Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Treball Final de Grau

Grau en Enginyeria Informàtica (GEI)

Enginyeria de Computadors

Improving nanos6 dependency subsystem

David Álvarez Robert
(david.alvarez@bsc.es)

Directors:
Vicenç Beltran Querol (vbeltran@bsc.es)
Eduard Ayguadé Parra (eduard@ac.upc.edu)

Computer Architecture Department (DAC)

Barcelona, June 2019

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Abstract

The OmpSs-2 programming model extends the basic dependency model of OpenMP with advanced features such as weak dependencies and dependencies over non-contiguous memory regions. These features also work across different task-nesting levels, providing a general and fine-grained synchronization mechanism. In general, these advanced features simplify the parallelization of complex applications and also improve performance. However, the current implementation of the data dependency subsystem in the Nanos6 runtime is complex and thus difficult to optimize for certain use cases. In this work, a new alternative implementation of the dependency subsystem aiming to provide a subset of the current features but focusing instead in performance and simplicity is designed, developed and evaluated. Additionally, a mechanism to switch between both implementations at execution time is also added to Nanos6.

Resum

El model de programació OmpSs-2 amplia el model bàsic de dependències d'OpenMP amb característiques avançades com dependències *weak* i en regions de memòria no contigües. Aquestes característiques també funcionen a través de diferents nivells de parentesc entre tasques, proporcionant un mecanisme de sincronització general i de gra fi. En el cas general, aquesta funcionalitat avançada simplifica la tasca de paral·lelitzar programes complexos i en millora el rendiment. No obstant això, la implementació actual del sistema de dependències a la llibreria Nanos6 és complexa i per tant difícil d'optimitzar per a certs casos. En aquest treball es dissenya, desenvolupa i avalua una nova implementació alternativa del sistema de dependències, amb l'objectiu de proporcionar un subconjunt de les característiques actual però centrant-se en el rendiment i la simplicitat. Addicionalment, s'incorpora a la llibreria un mecanisme per canviar entre ambdues implementacions en temps d'execució.

Table of Contents

List of Figures

List of Tables

1 | Introduction

Parallel programming is a difficult task. The process of transforming a sequential program into a parallel one is not straightforward and, even then, achieving the maximum degree parallelism and scalability for a given problem (and thus, the maximum performance) is even more difficult. As it poses such a challenge, several programming models exist with the goal of making this task easier to tackle.

Those programming models often offer different levels of abstraction. In the lowest abstraction level we find the *pthreads* model, part of the POSIX.1 Standard [1], and in the highest we find the domain-specific languages or DSLs. In between, there are other models which aim to provide a balance between ease of use, generality and performance, such as Cilk, Intel TBB, OpenMP, and the OmpSs family [2–5].

1.1 Motivation

This work focuses on the OmpSs-2 programming model [5], developed by the Barcelona Supercomputing Center (BSC). This model features a data-flow execution schema, in which the programmer adapts the software by using compiler annotations to split the code into tasks and indicating for each task on which input and output data structures it operates. Then the runtime system, based on the user provided annotations, executes the program in parallel if it can guarantee that the result would be equivalent to a sequential execution of the same program.

The data-flow execution model will ensure that the final result is correct by using the information provided by the programmer about the tasks to schedule them in such a way that doesn't break the constraints of a sequential model, while providing as much parallelism as possible. This is done through the concept of data dependencies inside the OmpSs-2 runtime, called Nanos6. The runtime calculates the dependency graph through the inputs and outputs declared by each task, and allows parallel execution of all the tasks that can be scheduled without breaking the sequential model.

The section of the Nanos6 runtime that calculates the dependencies and allows or holds the execution of the different tasks is the dependency subsystem, and it is one of the most important components of the runtime, as it is responsible for the parallel execution of the tasks and must ensure the equivalency of that execution to a sequential one, and thus forms the core of the data-flow execution model.

However, the dependency subsystem already implemented in the Nanos6 runtime is very complex, because it has to support all the features provided by the OmpSs-2 specification for the task construct, which are a lot since it is the main part of the model, and some programs do not need such a complete set of features as they have simpler data models.

This work focuses on the implementation of a simpler dependency subsystem for the Nanos6 runtime with less features than the existing one, but achieving better performance/scalability for certain task granularities. Additionally, the users of the runtime will be able to select which implementation of the dependency subsystem they want to use for different executions of their programs without the need to recompile all the libraries nor their programs.

1.2 Actors

The following roles will be needed during the development of this project.

- *Developer*: Responsible for writing the implementation that has been designed and agreed by both the developer and the director of the project. The Developer will also be in charge of writing the final thesis, designing the project plan, code analysis, coding, project management, research and documentation. In this case, it's the student doing the college thesis.
- *Support Staff*: Responsible of helping the developer carry out all tasks in the project's scope. In this case, the staff is the BSC's Programming Models team which works or has worked with the runtime and their experience will be of high value to the developer.
- *Directors*: They will supervise the developer, mainly through face-to-face meetings and emails. They will also have a strong influence in the technical decisions made during the implementation.
- *Users and beneficiaries*: The beneficiaries of this project will be the users of the Nanos6 runtime, which will be staff of the BSC, *MareNostrum* users that use OmpSs-2 and anyone in the world that chooses to download the GPL-licensed version of the runtime [6]. Also, this code may serve as a future reference for newer versions or different parallel programming models, as it is open-sourced.

2 | State of the art

This chapter provides context for the project by introducing related work through history and detailing the current state of the matter.

2.1 Parallel programming models

Parallel programming was born with the first supercomputers, that solved the need for more performance by having several CPUs, which would allow to speed up software as described by *Amdahl's Law* [7]. As such, High Performance Computing was the main motivation behind the first parallel programming languages and models, and it allowed for asynchronous programming in single-core CPU machines as well.

During the 1990s, several models were created, but one of the first ones to gain relevance was MIT's Cilk [2], based on the concept of *Work Stealing* [8]. Cilk was the first runtime for parallel programming that guaranteed near-optimal performance, strict correctness and predictable runtime for writing parallel software. It was based on a thread graph model where when one of the threads ran out of work, it would *steal* a task from another. It ran on the supercomputers of the era and, most importantly, it served as inspiration for future shared-memory based parallel models.

Shortly after some other standards such as OpenMP [9] and Intel Threading Building Blocks [3] appeared, based on the same shared-memory communication mechanism and the idea of annotating a sequential program to exploit parallelism while producing the same results as the sequential version.

2.1.1 OpenMP

OpenMP is a case worth studying, because it has become a standard in writing parallel programs. It was based on a *fork-join* execution model [4] that allows the programmer to define regions in the program that will be executed in parallel, and everything can be done by annotating the source code (in the case of C, with *#pragma* directives). The idea to annotate a sequential program, that could still be valid without annotations, and turn it into a parallel one, has been one of its main selling points.

Later, with the introduction of the *task* construct and dependencies, it also supported a different *data-flow* execution model, where no explicit synchronization mechanisms were needed.

There has been, however, some criticism during the years, specially regarding the missing support for tasks at the start, and the non-asynchronous parallelism focus, that have been enough to cause the creation of other programming models during the years.

That being said, major compiler support for OpenMP [10] is outstanding, including: GCC, Clang, Intel, IBM and more, so it is very established in the industry.

2.1.2 OmpSs and OmpSs-2

OmpSs, its successor OmpSs-2, and the StarSs model family in general, have been some of the fore-runner programming models to the OpenMP standard [5]. Their main goal is to be a testing ground for innovations and new concepts in the parallel programming research. Concepts introduced first on this programming models have later been adopted by the OpenMP standard, as explained in Subsection 2.1.3.

The StarSs family was created and is still being developed at the Barcelona Supercomputing Center. They feature only a *data-flow* execution model, simplifying over the OpenMP model but inheriting the concept of transforming programs through compiler annotation. Tasks are the main units of work in OmpSs and OmpSs-2, and they are ordered into execution to guarantee sequential equivalency thanks to the input and output data specified by the programmer.

Another of the main focuses of OpenSs-2 is the ability to offload work to different architectures and devices, transferring execution of tasks to GPUs and FPGAs in a way that is transparent for the programmer, becoming effectively a heterogeneous model.

2.1.3 Influence in OpenMP

The OpenMP standard is alive and has added during its lifetime many features that have been implemented before on other programming models. Specifically, many of the features that have been proposed and developed at the Barcelona Supercomputing Center for its OmpSs models have been later added to the standard, to see more mainstream use. Figure 2.1 shows some of the features that have been added to some versions of the OpenMP standard after being implemented first on OmpSs or its predecessors.

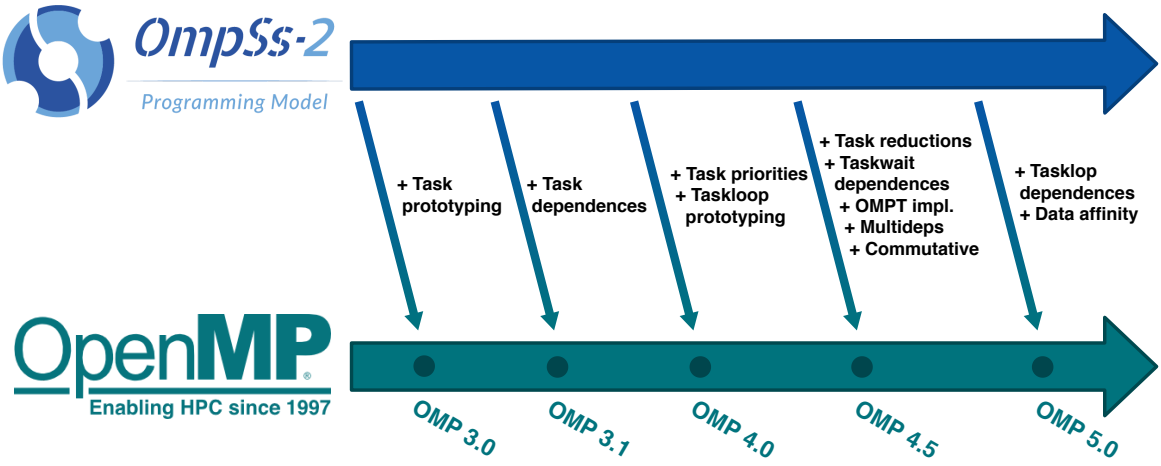


Fig. 2.1: OmpSs-2 features that have influenced the OpenMP standard.

The work done at the BSC on the OmpSs-2 model and its runtime, Nanos6, can be understood as experimentation on features that may be later incorporated to the standard if they prove to be able to be efficiently implemented and provide measurable advantages.

2.2 Data dependencies

Parallel programming models often require the programmer to specify what data is going to be accessed during a task, and what type of access will be done (read, write, both...). This is so a dependency graph between the tasks can be calculated, and the runtime can ensure the final result will be the same than a sequential program. If the same data is used in two sibling tasks, those two tasks have a **data dependency** between them.

In particular, we can distinguish three types of data dependencies that can cause race conditions [11]:

1. **True dependence (RAW)**. Read after write. Happens when a task T_1 outputs data that is going to be read by a task T_2.
2. **Antidependence (WAR)**. Write after read. Happens when a task T_1 reads data that is going to be overwritten by a task T_2.
3. **Output dependence (WAW)**. Write after write. Happens when a task T_1 writes data that is going to be overwritten by a task T_2.

Altering the order of operations in any of the above cases can cause a data race and hence an incorrect result in the execution of a program. Thus, any parallel programming model that does data-flow execution needs to be able to know when a RAW, WAR or WAW dependency can happen, and ensure correct synchronization to prevent races.

OpenMP supports a simple data dependency model through its *task* construct [4]. It allows for discrete dependencies between tasks at the same level of nesting, and the runtime will ensure correct execution constraints for those dependencies, by calculating a dependency graph and identifying potential data race conditions.

Other models support more complex dependencies, which allow the runtime to take more informed decisions about the task ordering, because they allow the programmer to specify more precisely the nature of those dependencies in the code. OmpSs-2 is one of those models [5], as data dependencies are the main mechanism to order task execution.

2.3 OmpSs-2 dependency model

OmpSs-2 extends the discrete dependency model of OpenMP to allow more fine-grained control through the *weakin* and *weakout* constructs [12]. In OpenMP's model, if task nesting is used, two tasks with different parents cannot be directly linked through a data dependency, because they will only work with tasks on the same level. Instead, the programmer is forced to define the dependencies between the parent tasks, even if it is really the child tasks having the dependency.

This can cause a performance penalty, because at some point there may be tasks that could have started the execution and are waiting due to the dependencies not being fine-grained enough.

In Figure 2.2 it is clear that Task 2.1 could be executed before Task 1.2 finishes, because they have no real dependencies. This would result in a performance penalty in a model without the *weakin* and *weakout* clause. These two clauses are used between the parent tasks (T1 and T2, for instance) and tell the runtime that those two tasks have a dependency that is not caused directly by them, but by one of their child tasks, that will make them explicit with the *in* and *out* constructs. With this, the runtime has enough information to achieve a better performance, potentially.

In fact, even parent tasks that are dependent through *weakdepend* clauses can be executed concurrently, because the runtime has the knowledge that as long as their child tasks are not executed, the result will be correct because the parents have no real dependency.

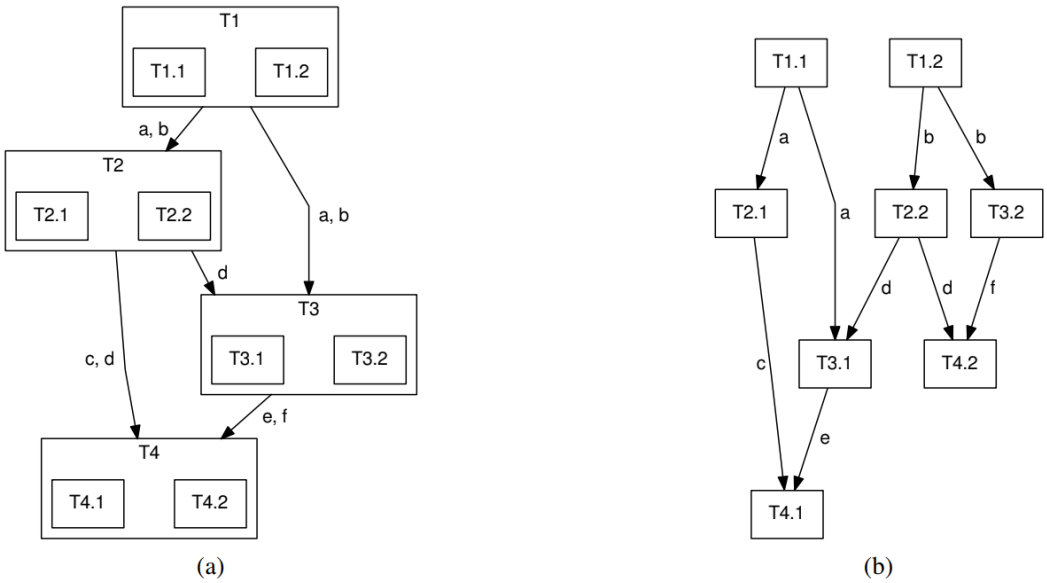


Fig. 2.2: Data dependencies without and with the weakin/weakout constructs

Another particularity of the OmpSs-2 dependency model is that the user can specify dependencies for sections of memory (that do not need to be contiguous), not just discrete points, and thus the runtime can track the dependencies by regions and have more parallelism for some user cases. In figure 2.3 there are examples of the OmpSs-2 API for the different dependencies.

```
1  int main () {
2      int a[500];
3      int b[500];
4
5      // Discrete dependencies without weak constructs.
6  #pragma oss task in(a) out(b)
7      doSomething();
8      // Discrete dependencies with weak constructs.
9  #pragma oss task weakin(a) weakout(b)
10     doSomethingElse();
11     // Region dependencies
12 #pragma oss task in(a[0:499]) out(b[0:249])
13     anotherComputation();
14 }
```

Fig. 2.3: Examples of an OmpSs-2 annotated program

The OmpSs-2 spec also includes support for reductions in the *task* construct [5, 13], which allow the programs to combine or accumulate results obtained in different tasks

without the need for any explicit synchronization mechanism, such as atomics or barriers, increasing performance. The runtime suppors *weakreductions* as well, which enable reduction nesting. Reductions are seen by the runtime as just another type of data dependency, and they are registered as such through the API.

Figure 2.4 showcases a simple use of a task reduction to combine results of different tasks without the need to explicitly synchronize access to data.

```
1   int a[500];
2
3   int main () {
4       /* A simple parallel sum of an array */
5
6       int sum = 0;
7
8       for(int i = 0; i < 500; ++i) {
9           #pragma oss task in(a[i]) reduction(+: sum)
10          sum += a[i];
11      }
12  }
```

Fig. 2.4: Example of OmpSs-2 reductions

A fully functional implementation of the OmpSs-2 spec for data dependencies already exists, but because of the great amount of features supported, specially memory region dependencies, that implementation has a very high degree of complexity and the overhead introduced might have a performance impact for some tasks with small granularities. The objective of this thesis is to develop a new implementation with less features that will have better performance for some applications.

3 | Scope

3.1 Goal

The main goal of this project is to design and write a correct and efficient implementation of as much of the Nanos6 dependency subsystem as possible, applying parallel techniques, algorithms and architecture-aware programming that have been learned during the course of the student's Informatics Engineering degree.

3.2 Requirements

- Design and code a correct implementation of the Nanos6 dependency subsystem.
- Adapt the code to the standards and style of the Nanos6 project.
- Test and verify the correctness of the implementation.
- Achieve better performance than older implementations in as much cases as possible.
- Implement the maximum of dependency features without compromising that performance.
- Study and evaluate the final result and suggest future work with the knowledge gathered during the project.

3.3 Scope

At the beginning of the project, existing implementations will be analyzed by the developer to gain knowledge about the runtime and understand what and how needs to be changed, to discuss with the directors the nature of the changes to the runtime.

During this phase, as much tests as possible will be written, to stress the runtime in different ways, and existing benchmarks will be prepared to be ran by the developer as well. This has two goals: to understand the use of the programming model from a

user point of view and to use those tests further down the line to verify the correctness of the new implementation. Both tests that use only one feature and tests that use multiple features in conjunction will be coded.

Then, the second phase will be related to prepare the runtime to have two or more different dependency subsystem implementations at the same time, and the user being able to dynamically select one at run time. This is needed because the new implementation might not be feature-complete enough to be used in all of the user cases, and thus it is not a drop-in replacement.

The next phase will be to design and code the new implementation, possibly based on earlier versions but with the goal of being as simple as possible to start with, allowing for more complex enhancements down the line. During this process, the tests mentioned earlier will be executed at each step to verify the progress and prevent regressions.

Finally, an evaluation will be carried out, with many different benchmarks and on different systems, to get a general understanding of how the new version performs and its limitation versus old implementations. At last, after the approval from the directors, the thesis will be written.

3.4 Risks

3.4.1 Deviation of the project plan

If the chosen design performs worse than expected, the project timeline or cost might change in order to accomplish the project goal.

Should this happen, the possibility of expanding the timeline would have to be explored, but this scenario can be prevented by doing a good job in the planning stage.

3.4.2 Introducing bugs to the runtime

Bugs are an integral part of software development, and they are caused by human limitations. In this case, difficult to reproduce bugs could add complexity to the project and endanger the project plan.

Software correctness cannot be totally guaranteed, but most bugs can be detected early using established and proven development methods, such as Test Driven Development [14], to detect them as soon as they enter the code. This is further explained in Subsection 5.1.3

3.4.3 Compilation and test times

The full Nanos6 runtime takes several minutes to compile. Even if only one file has changed, linking time is not negligible, and even then running all the tests and stressing the runtime will take time out of actual development.

The development method can be adapted to the situation by trying to compile fewer libraries of the runtime and do other tasks during compilation, but time will be lost due to this risk specially when doing quick troubleshooting.

3.4.4 Result variance

As there are infinite use cases for the runtime, some programs might perform better than others, and achieve different speedups.

Tests cases and performance benchmarks will feature as much variety as possible to ensure most use cases are being evaluated, not just a few.

3.4.5 Debug difficulties

With parallel software with big degrees of concurrency and other complexities, deadlocks or race conditions can be difficult to reproduce, find and solve.

The developer will have to learn how to use debugging tools in those cases and write specific tests to trigger edge-case behavior.

4 | Methodology

4.1 Time management

To use the time as efficiently as possible, tasks in this project will be short and incremental, having between them working versions of the software. This is similar to the SCRUM philosophy, but other concepts of that methodology will not be used because the project will not be done in a team environment. By doing it this way, with *Sprints* of one week or two, the planning can be changed on the go and that will provide a lot of flexibility with timing and planning [15].

4.2 Progress tracking

To keep track of the work that has been done, weekly (if possible) meetings will be held with the director, and the status updated online on *OneNote*.

For code tracking, on the other hand, the *Git* version control system will be used, in tandem with the *GitLab* online portal that is already used by the Programming Models group at the Barcelona Supercomputing Center, both to monitor progress and to exchange patches with other developers if needed.

4.3 Validation

Using a Test Driven Development approach, that is materialized because tests will be written before a single line of runtime code, will help correctness to be ensured during all steps of the project. That way, the implementation can also be defined as functionally complete when all the tests pass.

For performance, a set of benchmarks will be prepared to be run on different systems to gather speed-up statistics, and will also help finding bugs.

Finally, the final validation of the project will be done by the directors and the Barcelona Supercomputing Center.

4.4 Final result

The result expected for this project would be to get the code that has been developed merged into the master branch of OmpSs-2, reaching a release so that all the runtime users can access a better performance for smaller task granularities. The performance gain shall be quantified and proven through benchmark results in this thesis.

5 | Project plan

This chapter describes the different tasks that define the project, the dependencies between them and the resources that will be used. All of this will be illustrated with a Gantt diagram in Section 5.6.

Finally, the possible setbacks that can be encountered during the project will be discussed, in particular how they would affect the plan and the resources, and how they can be worked around if need be.

5.1 Tasks

The following subsections specify the different tasks that comprise the project.

5.1.1 Project management

This tasks encompasses all the content of the *Gestió de Projectes* (GEP) subject, and its different deliverables. In total, five different tasks will be delivered totaling 75 hours of dedication, distributed the following way:

- **Context and Scope**: 24.5 hours
- **Project plan**: 8.25 hours
- **Economical plan and sustainability**: 9.25 hours
- **Specialization module**: 12.5 hours
- **Oral presentation and final document**: 18.25 hours

The tools *Google Drive, Microsoft Office, Gantter, Google, Atenea* and *El Racó* will be used during the course of the subject.

5.1.2 Runtime analysis

Before starting to write new code, it is very important to analyze the code that is already there, and what the rest of the system expects out of it. This will provide an understanding of what needs to change and help prevent bugs down the road caused by misunderstanding what the code really does.

This task will also encompass the time the developer will need to become familiar with the compiler, debugging tools and environment needed for the runtime.

The following tools will be used during this task: *Git, GitLab, Visual Studio Code, Vim, CLion, Mercurium, GCC* and *Bash*.

5.1.3 Correctness tests

As has been mentioned in earlier sections, tests will be created to ensure correctness.

The tests will be small and totally independent programs that will use the OmpSs-2 tasks and dependencies, each one in a different way, to get as much coverage as possible. Each program will initialize the data, then do some parallel computation, and finally verify the result sequentially. High number of iterations, excessive task granularity and other techniques will be used in the tests to stress the runtime as much as possible.

The tests will be easy to run as well, so they can be ran automatically each time the runtime is compiled to verify no bugs were introduced.

To create the tests *C* will be used as the programming language, the *Mercurium* compiler and a code editor.

5.1.4 Initial implementation

Next, an initial implementation of the dependency subsystem will be designed and written, either from scratch or based on an already-existing one. The focus will be to keep it simple so optimizations and features can be introduced without a lot of development effort.

C++ will be used as the implementation language, compiled with the *GCC (GNU Compiler Collection)*, and any of the code editors available.

5.1.5 Optimization

The optimization of the runtime consists on identifying the bottlenecks and addressing them one by one. The developer will have to find the bottlenecks and come up with creative and simple ideas to solve them without introducing great amounts of complexity. The goal will be to enhance the initial implementation, both with new features and better performance.

C++ will be used as the implementation language, compiled with the *GCC (GNU Compiler Collection)*, and any of the code editors available.

5.1.6 Evaluation

With the correctness tests, explained in 5.1.3, and benchmarks readily available at the BSC or sourced from elsewhere, the performance gains of the new dependency subsystem versus the original will be evaluated objectively.

The benchmarks may be run at some of the different supercomputers available at the BSC, and to make the task easier, an automatic build and run script will be created, to collect and save the execution times and results of the different benchmarks.

The benchmarks implementation language may be *C++* or *C*, they will be compiled with *Mercurium* and any scripts that need to be written will be done in either *Bash* or *Python*.

5.1.7 Thesis writing

Finally, with all the research, knowledge and results gathered during the project, the final thesis will be written and future work may be proposed.

To write the thesis, LaTeXwill be used, with *Visual Studio Code* as the editor, *pdflatex* as the compiler, and any web search engines, books or sources that are needed.

5.2 Timing

Table 5.1 shows the estimated time to do each of the tasks specified in section 5.1.

Task	**Time (h)**
Project management	75
Runtime analysis	32
Correctness tests	32
Initial implementation	126
Optimization	157
Evaluation	32
Thesis writing	101

Table 5.1: Estimated time in hours required to do each of the tasks.

5.3 Task dependencies

Some of the tasks that have been defined in 5.1 require other tasks to be completed before they can be started. Table 5.2 shows prerequisites for each task, if applicable.

Task	Prerequisite
Project management	-
Runtime analysis	Project management
Correctness tests	Project management
Initial implementation	Runtime analysis Correctness tests
Optimization	Initial implementation
Evaluation	Optimization
Thesis writing	Evaluation

Table 5.2: Prerequisites for each task.

5.4 Resources

Different types of resources will be needed for this project, mainly human, hardware, and software.

5.4.1 Hardware

A laptop will be provided by the Barcelona Supercomputing Center, as well as a screen, for the purpose of this project. Also, the developer may work at home with a more-powerful home PC for any reason. Finally, more than one supercomputer may be used to check performance and correctness, but only the CTE-KNL supercomputer is specified here, as it will be the main resource used during the final evaluation task.

For each system, CPU and memory available is specified, as well as any other notable features.

- **BSC Laptop**
  - **CPU**: Intel® Core™ i7-5600U (2 cores, 4 threads, 2.6GHz)
  - **Memory**: 16GB
- **Home PC**
  - **CPU**: AMD® Ryzen™ 7 1700X (8 cores, 16 threads, 3.8GHz)

  - **Memory**: 16GB
  - **GPU**: NVIDIA GTX1060

- **CTE-KNL Supercomputer**:
  - **Login nodes**: CTE-KNL has 1 login node with the following configuration.
    * **CPU**: Intel ®Xeon™ E7-8850 (80 cores, 8 NUMA nodes)
    * **Memory**: 2TB
    * **Interconnect**: GPFS 10GBit/s (fiber)
  - **Compute nodes**: CTE-KNL has 16 compute nodes with the following configuration.
    * **CPU**: Intel ®Xeon Phi™ 7230 (64 cores, 256 threads, 1.30 GHz)
    * **Memory**: 96GB of main memory (90 GB/s), 16 GB of high bandwidth memory (480 GB/s) in cache mode
    * **Interconnect**: 100 Gbit OmniPath interface, GPFS 1Gbit

5.4.2 Software

This project will use a wide variety of software for the different tasks, as well as many libraries and such required for the runtime to compile. For this reason, only the main software is specified in this section, but other programs and tools with no cost associated will be used during the project.

- **KDE Neon 16.04**: GNU/Linux distribution on the student's Home PC.
- **Manjaro Linux**: GNU/Linux distribution on the BSC laptop. It was chosen because of its bleeding-edge versions of tools and compilers.
- **Microsoft Windows 10**: Used for the GEP subject.
- **Git**: Decentralized version control system, used for code and the final thesis writing.
- **GitLab**: Git repository management server, already used by the BSC.
- **GDB**: the GNU Project Debugger, used to find and troubleshoot bugs in the runtime.
- **Mercurium**: source-to-source compiler, used in the BSC, to transform OmpSs-2 annotated code into valid C/C++.
- **GCC**: the GNU Compiler Collection, both used to compile the nanos6 runtime and by back-end of the Mercurium compiler.
- **C, C++, Bash and Python**: Programming languages used.

- **Vim**: Console-based text editor.
- **Visual Studio Code**: Text editor.
- **CLion**: C/C++ Integrated Development Environment.
- **LaTeX**: Computer typewriting system used to write the final thesis.
- **Microsoft Office**: Office package used for the GEP subject.
- **Gantter**: Online service used to create Gantt diagrams and do general project management.

5.4.3 Human resources

As was explained more in depth in Section 1.2, this project will count with a director and co-director, support staff, and the main resource that will be the developer. In this case, the student doing his final thesis.

5.4.4 Spaces

The developer will work from a desk at the BSC, which will allow for direct contact with other runtime developers, and will make it easy to have weekly meetings with the project director.

5.5 Workarounds and action plan

In the hypothetical case of deviations to the project plan, the workaround would be different if it is just a setback in the normal development workflow, for example not finishing one of the tasks planned for a week of development, or a bigger setback that endangers the project plan.

In the event of a setback inside the normal development workflow, because of unexpected complexity, bugs, personal problems, etc., the planning can be adapted because of the flexibility that our Agile development model and week to week planning give. As such, it could be rethought how to spend the remaining time and the requirements for the next development cycle can be changed.

In case of something major, that affected possibly the hole project, for instance bureaucracy problems or one task becoming much bigger than expected, time could be cut on the optimization and features included in the runtime, at the cost of less performance / functionality than expected, but that would be addressed in future work if need be. That way, further effects on the project plan would be prevented.

5.6 Gantt diagram

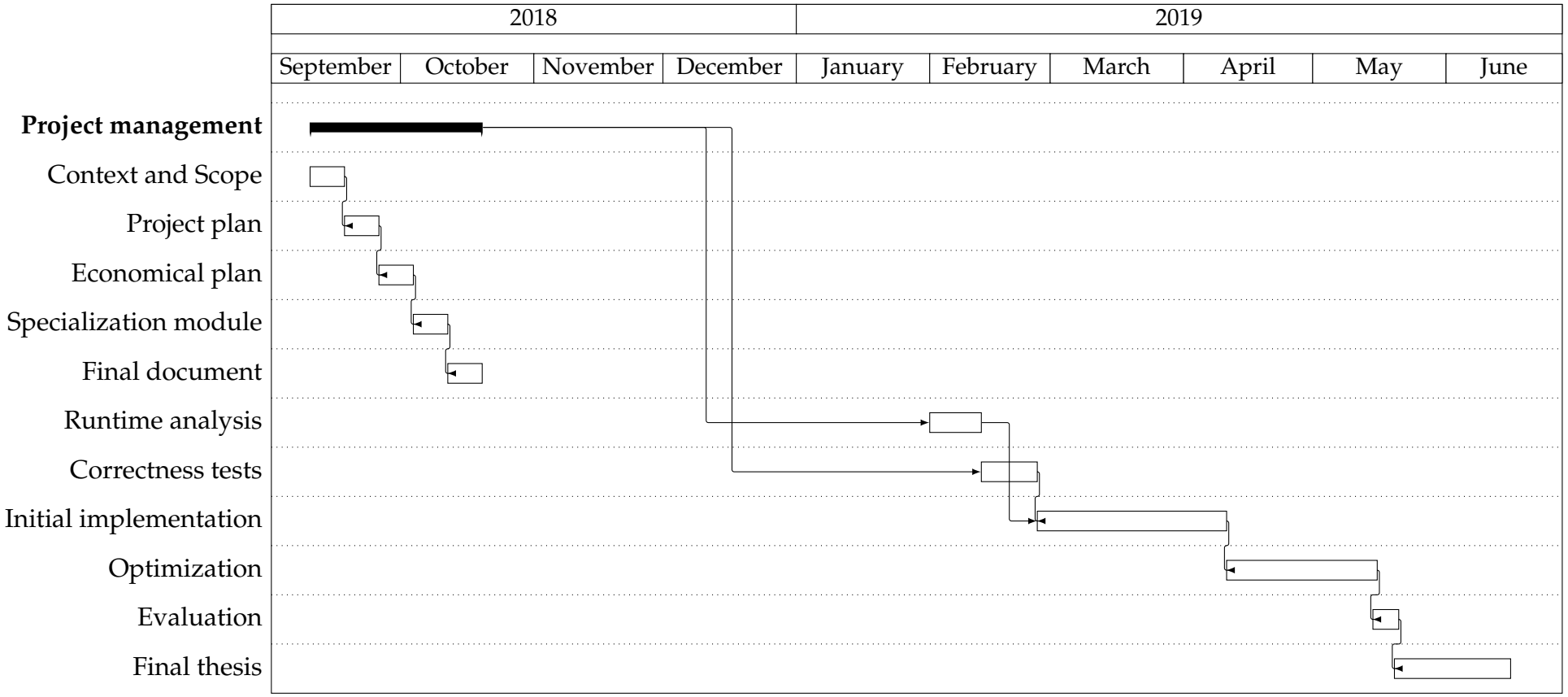


Fig. 5.1: Gantt diagram of the project timeline

5.7 Deviations

The Gantt diagram presented in Section 5.6 represents the final timeline of the project. On the initial plan, the student was due to start working at the BSC on October, but several bureaucratic issues caused this not to be possible. This was due to a change in the BSC policies for visitors, and it took a lot of time of coming and going between the University and the center. When all those issues were finally dealt with, it was possible to start working on the project in February.

Aside from that delay in the starting date, the rest of the project plan has been fairly respected. Although the exact count of hours spent in the project is difficult to calculate, the dates that were estimated for the start and end of the different tasks have been accurate and there was no rush at the end to finish the thesis in time.

6 | Economic management

This chapter breaks down all the costs related to the project and how to tackle possible deviations in the budget.

6.1 Direct costs

Direct costs are derived directly from the project. In this case, it is the human resources, the hardware and the software needed.

6.1.1 Human resources

The cost of labor for a Informatics Engineering student is approximated using the minimum wage of a student cooperation contract for the UPC as reference. As such, a cost of 8€/h is assumed. The hours of the BSC support staff invested in the student are added up as well, which have been estimated in 16 hours at market price.

Resource	**Price (€/h)**	**Amount (h)**	**Total (€)**
Informatics Engineering student	8	555	4440
BSC Support staff	20	16	320
Total			4760

Table 6.1: Cost of human resources.

6.1.2 Software

The price of the software will be calculated estimating the useful life of the licenses to obtain a price per hour, that can then be factored in for each task.

It is worth mentioning that most of the software in this project is free and open source, and thus it has no cost. This software will be omitted in the final budget for brevity, but is displayed here.

Resource	Price (€)	Life (years)	Amortization (€/h)
KDE Neon, Manjaro	0	-	0
Windows 10 (Student)	0	-	0
Git, GitLab	0	-	0
GDB, GCC, Mercurium	0	-	0
Vim, VSCode	0	-	0
CLion (Student)	0	-	0
LaTeX	0	-	0
Microsoft Office	150	3	0.026
Gantter (Trial)	0	-	0
Total			0.026

Table 6.2: Cost of software resources.

6.1.3 Hardware

There are two main hardware resources: the BSC laptop and the home PC. The amortization of the CTE-KNL supercomputer is calculated as well, but based only in one node (as only one at a time will be used during the project). However, its use is estimated at 12h/day, as the supercomputer has a lot of users inside and outside the BSC.

Resource	Price (€)	Life (years)	Amortization (€/h)
Home PC	1200	4	0.16
BSC Laptop	2000	4	0.26
CTE-KNL (node)	10000	4	0.57
Total			0.99

Table 6.3: Cost of hardware resources.

6.2 Indirect costs

Indirect costs are caused by the project but not as a direct cause of its activities. In our project, the main indirect cost is the electricity used by the spaces and computers, and the home internet.

The assumptions made for this cost are that the home PC uses about 400 W, the BSC laptop about 50W and, based on the CTE-KNL CPU's TDP of 215W, 500W for the full node accounting for cooling and other systems.

Resource	Price	Amount	Total (€)
Home PC Power	0.13847 €/KWh	400Wh * 75 h	4.15
BSC Laptop Power	0.13847 €/KWh	50Wh * 480 h	3.32
CTE-KNL Power	0.13847 €/KWh	500Wh * 40 h	2.77
Internet	40 €/month	5 months	200
Total			210.24

Table 6.4: Indirect costs.

6.3 Final budget

With the costs calculated earlier, and having an overall contingency of 10% and special budget for hiccups on the main tasks, the final project budget is presented in Table 6.5.

Resource	Units	Price (€/unit)	Total (€)
Project management			
Home PC	75h	0.16	12
Microsoft Office	75h	0.026	1.95
Student	75h	8	600
Runtime analysis			
BSC Laptop	32h	0.26	8.32
Student	32h	8	256
Correctness tests			
BSC Laptop	32h	0.26	8.32
Student	32h	8	256
Initial implementation			
BSC Laptop	126h	0.26	32.76
Student	126h	8	1008
Support Staff	8h	20	160
Setbacks	30h at 10%	0.8	24
Optimization			
BSC Laptop	157h	0.26	40.82
CTE-KNL	20h	0.57	11.4
Student	157h	8	1256
Support Staff	8h	20	160
Setbacks	30h at 20%	1.6	48
Evaluation			
BSC Laptop	32h	0.26	8.32

CTE-KNL	20h	0.57	11.4
Student	32h	8	256
Thesis writing			
BSC Laptop	101h	0.26	26.26
Student	101h	8	808
Indirect costs			
Home PC Power	75h	0.055	4.15
BSC Laptop Power	480h	0.007	3.32
CTE-KNL Power	40h	0.07	2.77
Internet	5 months	40	200
Subtotal			5203.79
Contingency	10%		520.38
Total without VAT			5724.17
VAT	21%		1202.08
Total			6926.25

Table 6.5: Final budget.

6.4 Risk management

Two different forms of risk contingencies are included in the final budget, just in case setbacks happen and threaten to deviate the costs.

First, in the two main development tasks, that have more changes of having hiccups because a lot of variables that affect those tasks cannot be accurately predicted, a 10% and 20% risk has been included, adding each risk a total of 30 hours of the student's time in case it is needed.

Second, a 10% contingency item has been added to the final budget. The percentage is low because it is not foreseeable that any other material would need to be bought in any setback scenario, and thus the project would only need more time and the corresponding software/hardware amortization for the extra work. This will cover any other incidents that may happen and have not been accounted for in advance, and ensure the budget is respected regardless.

7 | Sustainability

It is important to make an analytical reflection on the sustainability of the project to be able to justify its existence. This sustainability must not be just economical, but also social and environmental.

Table 7.1 represents the sustainability matrix of the project, where there is a brief summary of each of the elements and then a subjective score determined by the student. In the following sections on this chapter each of the elements of the matrix will be elaborated on.

	PPP	**Lifespan**	**Risks**
Environmental	74KWh of partly renewable energy + commuting	Reduction of emissions proportional to speedup	Deviation from project plan or low speedup
	8/10	10/10	6/10
Economic	6926.25 €	Efficiency increase proportional to speedup	Deviation from project plan
		10/10	8/10
Social	Learning and experience	Enhancement for users	None
	10/10	10/10	10/10

Table 7.1: Sustainability matrix and scores.

7.1 Economic dimension

The cost of the project is not too high for being a five month endeavor, but understandable as the main resource is a single student and almost no material is needed. In fact, the cost of the project to the BSC will be limited to just the space and a laptop, plus the hours of the director, as there is no internship involved.

As has been explained in Chapter 3, the focus of the project is to enhance the performance of the Nanos6 runtime. The runtime is currently used at the *MareNostrum*

supercomputer, and potentially many more systems around the world, as it is free software. Any improvement in performance will result in decreased software execution times, and time in a supercomputer is expensive. That way, it will cost less money to obtain the same results, because the software will be faster.

Aside from this, OmpSs has influenced the OpenMP spec heavily [5]. Many features that were first introduced in OmpSs have then been incorporated to the standard, and some of the reference implementations for those features were developed in the BSC. As such, the importance of having a good implementation is key, and it could decide if this features end up in OpenMP and benefit a much wider audience of users.

7.2 Social dimension

As an Informatics Engineering student, the main goal for the project is to learn. Being able to work side by side with the BSC and develop free software in a runtime used by other many developers, is a big learning opportunity and a great added value to the project. This is the main benefit the student will get from the college thesis.

The enhancement of a runtime will also affect directly to the quality of life of its users, now and in the future, as they will be able to write faster software retaining a lot of control over the data dependencies on their code. They will also gain confidence in the ability for Nanos6 to execute their code with good performance.

This new implementation also answers direct need of BSC staff, that need faster implementations with less features for certain cases, and simple code to be able to adapt to their needs easily.

7.3 Environmental dimension

All projects have an environmental impact, but the goal is to negate that impact with the benefits of the project during its lifespan. However, power used for the computers, and even fuel needed for commuting to and from the BSC, will create a footprint that cannot be ignored.

It will be a priority to reduce the environmental impact of the project as much as possible. This will include, among other things, using public transportation whenever possible for commuting. It is important to mention that the student's home uses electricity sourced on its entirety from renewable energy, which will help reduce the CO_2 emissions.

Once the modification to the runtime has been deployed, OmpSs-2 programs may have less execution time, and that is positive for the environment. A faster program is one that also, incidentally, uses less energy in order to execute. That will reduce the CO_2 footprint of the programs.

Another effect will be towards supercomputer amortization. As there are always programs running in the *MareNostrum*, the power bill is unlikely to be reduced, but if the software that runs on the supercomputer is faster, greater efficiency is achieved, more programs will be executed in the same amount of time. That is better for equipment amortization, not only economically, but environmentally as well.

8 | Analyzing the runtime and writing tests

This chapter covers the process of familiarizing with the current Nanos6 runtime and the coding of tests to check that different features related to the dependency subsystem behave correctly.

This starts all the way from the instrumentation and source to source compilation done by the *Mercurium* compiler, a high level overview of the OmpSs-2 runtime, reading the spec and user interfaces, and then writing valid C OmpSs-2 programs.

8.1 Compiling the runtime

The first step of this process will be to get a system ready to execute OmpSs-2 programs. It will be assumed that the *Mercurium* compiler is already installed on the system, as it is not the focus of this thesis. The importance of understanding the runtime compilation process will become apparent in Chapter 9.

The Nanos6 runtime uses the *GNU Build System* [16], also known as *Automake* to generate the different pieces needed for compilation. This allows developers to create makefiles and configure scripts for a wide variety of UNIX-like operating systems without having to hand-write all of them.

There are two important files for this process: the *configure.ac* file, which defines configuration options that the user will be able to specify when compiling the runtime, and the *Makefile.am* file, which is a higher-level Makefile that can use the flags defined in the configure script. All the files a developer wants to include in the final library must be added to that file.

The current compilation script includes an option to include different dependency subsystem implementations in the final binary. As there have been several iterations of the OmpSs-2 spec [5], several options can be chosen. However, due to the fact that there was a default value to be chosen and then the rest of implementations weren't even compiled, they were outdated to the point they couldn't be built anymore.

The rest of the compiling process is the standard for any UNIX software tarball. Running the configure script, compiling the source and installing.

8.2 Writing and compiling OmpSs-2 programs

The process to write and compile C programs that use OmpSs-2 is very similar to writing programs with OpenMP, but instead of using the *GCC* compiler, *Mercurium* is used. This section will walk through that process with a simple program, which without any parallelization, is shown in Figure 8.1.

```
1  int main(int argc, char *argv[])
2  {
3      for(int i = 0; i < 1000000; ++i) {
4          doComputation();
5      }
6      return 0;
7  }
```

Fig. 8.1: Simple C sequential program

Assuming doComputation() is expensive, this program may take a long time to execute. For the purpose of this example, it is assumed that all the calls to doComputation() have no data dependencies between them. As such, this program is parallelizable, and it can be annotated with OmpSs-2 *pragmas* to achieve parallel execution [5]. Figure 8.2 shows the same program annotated with OmpSs-2.

```
1  int main(int argc, char *argv[])
2  {
3      for(int i = 0; i < 1000000; ++i) {
4          #pragma oss task
5          doComputation();
6      }
7      return 0;
8  }
```

Fig. 8.2: Simple C parallel program

If the program in Figure 8.2 is compiled with the *Mercurium* compiler, it will be modified to dynamically load the runtime upon execution, and then execute the doComputation() calls in parallel. More information on the OmpSs-2 API is available in the spec [5].

8.3 Runtime architecture

The Nanos6 runtime is a complex system composed of different parts with a high level of integration and interconnection between them. As it has many components, and the developer is not familiar with many of them, only some are highlighted, and then the ones that are relevant for the project are explained in more detail.

- **Scheduler**: Distributes and assigns tasks (work units) between the different threads when they are ready to be executed. It will accept hints from other subsystems to decide what order the tasks are executed in.
- **Instrumentation**: Receives calls from all of the different systems signaling events and situations. Then, depending on the compile flags, it will use that information to print it, display it as a graph, pass it to other programs such as *Extrae* [17], or do nothing on the optimized version.
- **Loader**: Binary the annotated programs are linked with. Explained in detail in 8.3.1.
- **Dependency subsystem**: Keeps track of the data dependencies and decides what tasks are allowed to start execution. Explained in detail in 8.3.2

8.3.1 Loader

The loader is a small binary that is compiled with the Nanos6 runtime, and it is the one all annotated programs are linked against. Its main task is to then load the correct Nanos6 library depending on the environment variables the program has been executed with.

This is because the runtime has different *flavors*. Those flavors are essentially compilations of the runtime done with different flags that enable or disable certain verbosity, debug features, optimizations, etc. In its current form it is used by the user through *NANOS6* environment variable. Depending on that variable a certain flavor will be dynamically linked, and this way the user can enable different instrumentations.

Essentially, the loader is just a stub to link the real runtime, but it will link different flavors without having to recompile it. Some examples of the currently available runtime flavors are: verbose, debug, stats, stats-papi, profile, extrae and graph.

8.3.2 Dependency subsystem

The dependency subsystem is the one that will receive all the information regarding the task's data accesses, and then decide if that task can start execution, or it can't because it would violate the dependency model.

During the project no deep dive into the details of the current implementation was done, because it is of little value as it is going to change, and because it would probably influence the design of the new implementation. However, it is really important to understand the entry points to the system, both internally and externally, and what its responsibilities are.

To save its state, the dependency subsystem has a class called *TaskDataAccesses* that is a member of the main *Task* class. There, any relevant data structures are created to store the dependencies of the current task, and it will be allocated and destroyed with the task.

To maintain the state, the system has a series of functions that it exposes, either to the rest of components or even to the annotated program. They will be called in a specific order for each task, and allow the implementation to do any relevant operations on the data to ensure the dependency model is respected. Those functions are briefly explained here, and are called in the following order:

1. *nanos6_register_*_depinfo*: The * is substituted by the type of access (for example *write* or *read*), and this functions are exposed through the loader and called directly from the annotated program. One call will be made for each data access that task has declared.

2. *DataAccessRegistration::registerTaskDataAccess*: This function is called from inside the earlier one for each invocation, but is internal.

3. *DataAccessRegistration::registerTaskDataAccesses*: Called from the *nanos6_submit_task* function, only once per task, when all the accesses have been registered. This is where the dependencies and order are calculated.

4. *DataAccessRegistration::handleEnterTaskwait / DataAccessRegistration::handleTaskExitTaskwait*: Called when the tasks enters or exits a taskwait. Is is worth mentioning that implicit taskwaits (ones at the end of a task code block) also call this function.

5. *DataAccessRegistration::unregisterTaskDataAccesses*: Called when the data accesses of the task have been finished and dependencies can be satisfied.

6. *DataAccessRegistration::handleTaskRemoval*: Called whenever a task is being deleted from memory because it is not needed anymore.

To communicate to the rest of the runtime the status of the dependencies of a task, two counters in the *Task* class are changed:

1. *Task::increasePredecessors / Task::decreasePredecessors*: Marks the unsatisfied dependencies a task is pending on. When it reaches 0, the task can be scheduled.

2. *Task::increaseRemovalBlockingCount / Task::decreaseRemovalBlockingCount*: Keeps track on how many subsystems depend on that task remaining in memory to work, and hence are blocking the deallocation of the task. When it reaches 0, the task is deleted.

It must be noted that in the Nanos6 runtime, it is assumed that if a subsystem decreases to zero one of the counters of the earlier list, it is that subsystem's responsibility to either enqueue the task in the Scheduler or call the task destructor and deallocate it.

8.4 Writing correctness tests

With all the information obtained and after reading the OmpSs-2 Spec[5], the test writing phase can be started. They are written as simple self-contained C programs, that might or might not have a useful purpose, but that its correct execution can be checked easily in a sequential manner.

The coded tests check many different features of the dependency model but in isolation. To enumerate some of those features, there are discrete dependencies, task nesting, reduction, reduction nesting, weak tasks, totally overlapping regions, partially overlapping regions in different ways, etc. The main goal has been to cover as many cases as possible with the tests, to ensure that if a bug is introduced it breaks a test.

During all of the project, if a bug is found that doesn't break a test, a broken test just for that bug is created. After it, it is fixed, and checked with the new test that it is not introduced ever again. This is commonly known as regression testing [18].

In Figure 8.3 the typical layout of a correctness test is displayed. It sets up the needed data, does some work, checks the result, and then displays wether it was a success or a failure. Then, with a shell script, all tests can be ran and regressions are easily spotted. This specific test focuses on partially overlapping memory regions.

```
...
#define LENGTH 1024
#define TIMES 10

void sum_array(int* arr, size_t length, int from) {
        for(size_t i = from; i < length; ++i)
                arr[i]++;
}

int check_array_is_asc(int* arr, size_t length, int times) {
        for(size_t i = 0; i < length; ++i) {
                if(arr[i] != ((i+1)*times))
                        return 0;
        }

        return 1;
}

int main(int argc, char *argv[])
{
    int* arr1 = init_int_arr(LENGTH);
    int* arr2 = init_int_arr(LENGTH);

    for(int i = 0; i < TIMES; ++i) {
            for(int j = 0; j < LENGTH; ++j) {
                    #pragma oss task inout(arr1 [j:LENGTH-1])
                    sum_array(arr1, LENGTH, j);
                    #pragma oss task inout(arr2 [j:LENGTH-1])
                    sum_array(arr2, LENGTH, j);
            }
    }
    #pragma oss taskwait

    if(check_array_is_asc(arr1, LENGTH, TIMES) && check_array_is_asc(arr2
        , LENGTH, TIMES))
            printf("tasks_partial_overlap test 1: OK");
    else
            printf("tasks_partial_overlap test 1: FAIL");
    ...
}
```

Fig. 8.3: Example of a correctness test with tasks depending on memory regions with partial overlap

9 | Adapting the runtime for different implementations

This chapter explains how the runtime has been adapted, through a method of conditional compilation, to house two different implementations of the dependency subsystem that the user can switch without having to recompile.

The main purpose of this task is to be able to have both (original and new) versions of the dependency subsystem at the same time, as the next version may not have all the features the OmpSs-2 spec [5] requires, exchanged for performance in applications with low task granularity.

Please note, from this point onward, the original implementation will be referred as *linear-regions-fragmented*, which is its internal name, and the new implementation as *discrete-simple*.

9.1 Designing the mechanism

In the initial brainstorming phase, it had to be decided how to switch versions, and decide what technique was going to be used.

The first idea was to adapt the *linear-regions-fragmented* version to use the C++ version of an interface (an abstract class), that all the future versions could inherit from. That way versions can be switched at execution time, by having every current call to the dependency subsystem go through an implementation-agnostic interface.

That idea, however, had several issues that made it unsuitable. First, the time investment required to adapt the runtime in such a way was too much, as the Instrumentation subsystem relied on a lot of internal details of the *linear-regions-fragmented* version that could be easily mocked by other versions but was too time-consuming to abstract through interfaces.

The second issue, and the biggest one, was that all the data structures for the subsystem, housed in the *TaskDataAccesses* class, were allocated with the main task, and thus the size of those structures had to be known on compilation time. While decoupling them to be allocated on a separate call would be possible, memory allocation is

very expensive and all the references to that structure across the whole runtime would have to be changed as well.

Another idea was to use a similar conditional compilation method that was already used for the different Nanos6 flavors and has been explained in Section 8.3.1. That way, the existing loader is repurposed to switch dependency implementations depending on environment variables, and this idea has none of the issues of the first one, because the two versions would be compiled to different libraries.

The conditional compilation method was not free of drawbacks, though. By having to compile all the different Nanos6 flavors as well with each implementation, the number of binaries generated and the already long compilation time is doubled. That, however, was acknowledged as an acceptable trade-off and thus this approach was chosen.

9.2 Implementing conditional compilation

In Section 8.1 it was explained what tools were used to compile the runtime, and the relevant files. To get all the flavors to compile with both implementations, the *configure.ac* file was modified to provide switches to compile or omit each version. That way, if a user or developer is only interested in using one of the versions but not the other, they can be excluded from compilation, resulting in fewer binaries and lower compilation times.

The most important part, however, was to change the *Makefile.am* file and add definitions to build all the libraries twice, one for each version. That was done while trying to make it as easy as possible to add even more dependency implementations in the future, but the *Automake* tool has some limitations on its ability to define macros to be that portable (it doesn't even have loops, for example).

After the adaptations both versions could compile at the same time, but adapting the Nanos6 loader was still necessary.

9.3 Modifying the loader

The Nanos6 loader, explained in Subsection 8.3.1, allows the dynamic linking of different flavors by switching on environment variables. To be able to select the dependencies as well, a new environment variable *NANOS6_DEPENDENCIES* was added, with two possible values: *linear-regions-fragmented* and *discrete-simple*. Depending on that variable, the correct library will be linked to the annotated executable.

For that, the existing loader, which already had support for different flavors, was modified by adding the dependency implementation name to the loaded library as well, and if no environment variable was set, defaulting to the feature-complete *linear-regions-fragmented*, so the user had a spec-compliant library if nothing was specified. Figure 9.1 shows a code snippet of the new modification included in the loader.

```
void _nanos6_loader(void)
{
    ...

    // Flavor switch
    char const *variant = getenv("NANOS6");
    if (variant == NULL) {
        variant = "optimized";
    }

    ...

    // Dependencies switch
    char const *dependencies = getenv("NANOS6_DEPENDENCIES");
    if(dependencies == NULL) {
        dependencies = "linear-regions-fragmented";
    }

    ...

    // Load the library with name libnanos6-[variant]-[dependencies].so
    _nanos6_loader_try_load(verbose, variant, dependencies, lib_path);
    ...
}
```

Fig. 9.1: Loader code to switch flavors and dependencies.

10 | Initial implementation

This chapter covers the base design and code of the *discrete-simple* implementation. This was built based on another simple implementation that was developed earlier just to cover the needs of one particular use case. As the base code was adopted to cut down on development time, but does not represent the final design, details about it will be omitted.

10.1 Initial design

The basic idea of the implementation is that each task will have only two basic structures to keep its dependencies:

- A **Bottom map**, which will have entries pointing to the last access to a certain memory address. That way, when registering its dependencies, a child task can go to its parent's bottom map and determine if any of its siblings are accessing to the same memory addresses, and stablish a dependency relationship. The key of the map is the address. Inside one node of the bottom map, the following data is stored:
  - The **last access** that has been registered to that address.
  - A boolean indicating if that last access **is satisfied**.
- An **array of accesses**, that represent all the declared depend clauses of the current task. The actual addresses are stored in another array with the same indexes, to prevent long searches due to cache misses. For each memory access, the following information is:
  - The **access type**, being for now either read, write, or readwrite.
  - The **successor task**, which is the next sibling task that has registered an access to that address.
  - An boolean indicating if the access **is the top** of a chain of read accesses.

At one given point in time, there can only be one write/readwrite access to the same address, or n read accesses. This is the basic constraint the system will operate on.

Whenever a task registers its accesses, a counter described in Subsection 8.3.2 will be increased one time for each access that has dependencies blocking the task from starting execution. An access is only considered satisfied whenever one of this conditions is met:

- There is no bottom map entry for the address being accessed.
- The current access is a read, and the bottom map entry points to a satisfied read access.

Note that write accesses will always be unsatisfied when they have predecessors, because only one can be run at a time. Even when satisfied, the accesses will link their predecessors to them and change the bottom map entry.

When a read task is registered, its *top* bit will be set if the task is the first one (there is no bottom map entry).

When a task releases its dependencies, it has to iterate through its accesses and satisfy its successors. Different types of accesses will have different responsibilities as well. Whenever a access is released, the following things can happen:

- The task is a **read**.
  - If the task has the *top* bit set, it will mark itself as deletable and find its successor.
    - If there is no successor, it's bottom map entry will be deleted.
    - If the successor is a **write**, it will be satisfied.
    - If the successor is a **unfinished read**, the *top* bit will be set to true.
    - If the successor is a **finished read**, it will be marked as deletable, and the same logic will be applied recursively to the next successor.
  - If it is not the *top* read, it will not mark itself as deletable and will do nothing, because other reads are pending.
- The task is a **write** or **readwrite**
  - It will mark itself as deletable and satisfy its successor. If the successor is a read, all subsequent reads will be satisfied as well and the first one marked as *top*.

Note that tasks may not be deleted just as they release their dependencies, because in the model, accesses are an integral part of the dependency control data structures, and cannot be deleted until they are not needed anymore. This adds some more complex logic to the system, but is very efficient in terms of space.

To sum up the design, let's look at a simple yet real example. Figure 10.2 shows a dependency graph that corresponds to the sample code of Figure 10.1. Note that Task 1 has two Write accesses, to address A and B and drawn as circles, and Task 2 and

```
1   int main() {
2       int* A, B;
3       initialize(A, B);
4
5       #pragma oss task out(*A, *B)
6       {
7           do_calculation(A);
8           do_calculation(B);
9       }
10
11      #pragma oss task in(*A)
12      print_by_screen(A);
13
14      #pragma oss task in(*A)
15      output_to_file(A);
16  }
```

Fig. 10.1: Sample code of a simple write-then-read program

Task 3 have read accesses to address A, drawn as diamonds. The figure represents a snapshot of the structures just as all tasks have been registered.

A possible event timeline would go on in this example, following the design:

1. Task 1 is created. Accesses to A and B are registered. Since there are no entries in the bottom map for those accesses, they are created, with the satisfied bit set to true. Task 1 has no dependencies, so it can start.

2. Task 2 is created. Access to A is registered, and an entry is found in the bottom map. As the earlier access is a write, Task 2 cannot start. The bottom map entry now points to Task 2 and is not satisfied.

3. Task 3 is created. Access to A is registered, and an entry is found in the bottom map. As the earlier access is a **unsatisfied** read, Task 3 cannot start. The bottom map entry now points to Task 3 and is not satisfied.

4. Figure 10.2 represents this point in time.

5. Task 1 finishes. During its dependency release, it will satisfy Task 2, place it in the Scheduler queue, and mark A as a Top access. As Task 2 is a read, it will travel down the dependency chain, and satisfy any other read access, as they can be concurrent. Task 3 will be satisfied as well, and the bottom map entry marked as satisfied.

6. Task 2 finishes. As it was marked as top, it will pass down the ”Top” bit to the next unfinished access in Task 3, and will delete itself.

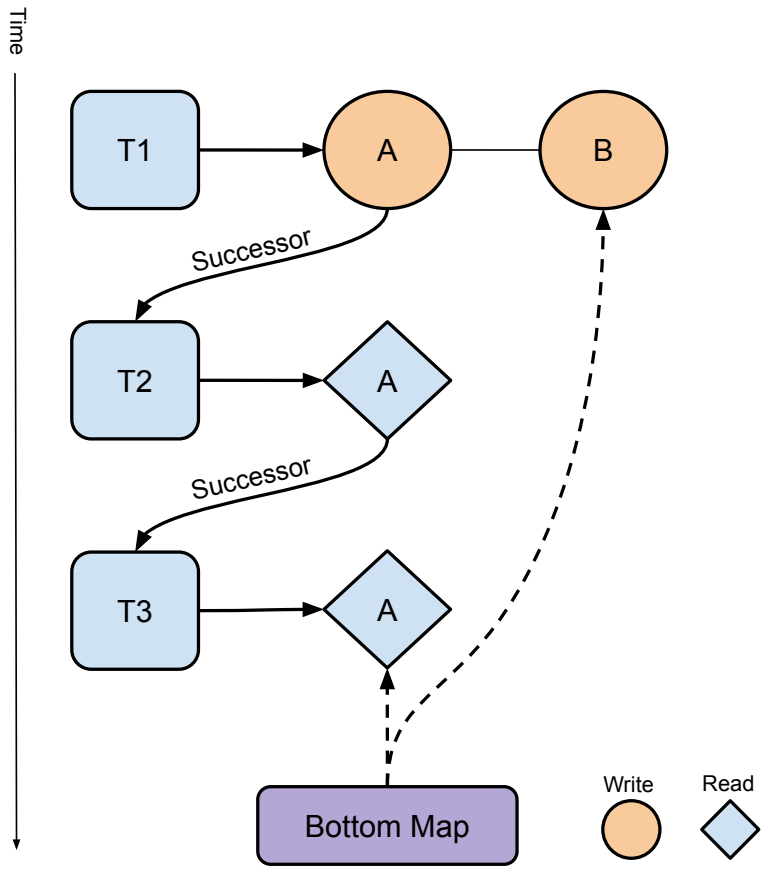


Fig. 10.2: Dependency graph of a simple write-then-read program

7. Task 3 finishes. As it is marked as top, will delete itself and the corresponding entry from the bottom map.

10.2 Task nesting

So far, it has been described how the system allows for dependencies between siblings, but nothing has been said about nesting. However, when the data structures were described, it was said that the bottom map is allocated on each task. That is because that map is only used by children of that task, and in the Nanos6 runtime even first-level tasks have a parent, called the *main task*.

It is also important that the OmpSs-2 spec [5] makes it mandatory to declare dependencies of child tasks at the parent level as well, either through the *depend* or *weakdepend* clauses. With that, task nesting already works in the implementation.

If Figure 10.4, corresponding to the sample code of Figure 10.3, is considered, it can be seen that through the already defined mechanisms, the dependencies will be respected. As dependencies are declared at parent level as well, Task 1 will always execute before

Task 2, and then the dependencies between the children of Task 1 will be allocated in the Task 1 bottom map, which was empty, and will be executed in order.

```
1   int main() {
2       int* A, B;
3       initialize(A, B);
4
5       #pragma oss task out(*A, *B)
6       {
7           #pragma oss task out(*A)
8               do_calculation(A);
9           #pragma oss task out(*A)
10              do_calculation(A);
11
12          do_calculation(B);
13      }
14
15      #pragma oss task in(*A)
16      print_by_screen(A);
17  }
```

Fig. 10.3: Sample code of a simple program with task nesting

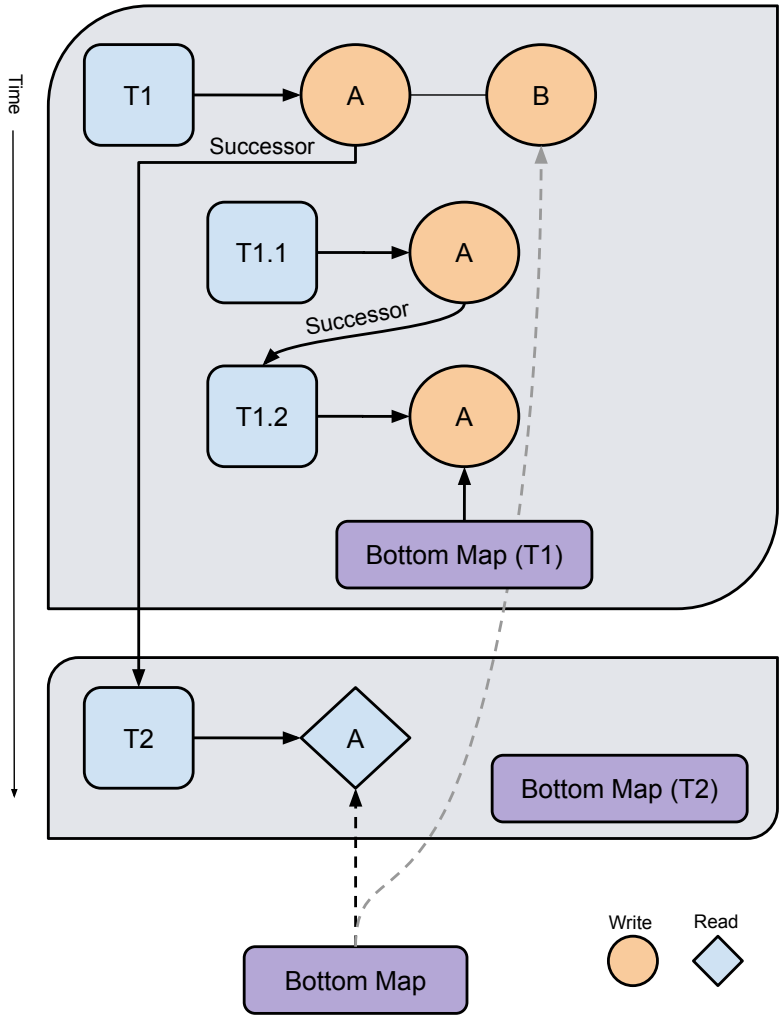


Fig. 10.4: Dependency graph of a simple program with task nesting

As there is an implicit taskwait at the end of each task, Task 1 will never finish before Task 1.1 or Task 1.2, and thus will never release its dependencies before their children finish.

This features even allow to enable support for *weakin* and *weakout* clauses by treating them as normal *in* and *out* clauses, altough none of the benefits from the early release mechanisms is obtained, and parent tasks will most likely be just serialized.

10.3 Implementation details

In this section, only non-trivial or particularly interesting implementation details will be discussed, and code examples will be included for clarity.

10.3.1 Locking

In a concurrent environment, each time a data structure is used it has to be considered if it is possible for multiple threads to access that structure simultaneously, and if explicit synchronization of those threads is needed.

For the bottom map structure, as it is implemented as a C++ STL *unordered_map* [19], it is only thread-safe for concurrent reads, and as such some sort of synchronization mechanism must be used. A lock that already exists and is used on the Nanos6 runtime will be repurposed for this case: a Ticket Spinlock, which is a ticket-based busy wait lock, designed to reduce contingency and prevent context-switching.

Every task will have its own spinlock for its bottom map, and all the child tasks will take the lock when reading or writing to it. As the implementation has been designed to touch only the bottom map when registering tasks and when the last access finishes, such coarse locking scheme should not be a problem.

This lock will also be taken to double-check when a task accesses its successor and finds there is none, as there could be another thread changing that at the same time. Figure 10.5 shows a code snippet of one of those double-check mechanisms.

This very basic one-lock scheme will contribute to the overall simplicity of the implementation, as there is no possibility to create deadlocks if just one lock is in use at any time.

```
1   if (successor != nullptr) {
2       ...
3   } else {
4       // Could be false positive. We need a lock on the bottomMap.
5       std::lock_guard<TaskDataAccesses::spinlock_t> guard(accesses.
            _lock);
6       if (pAccess->getSuccessor() == nullptr) {
7           // No race.
8           bottom_map_t::iterator itMap = accesses._accessMap.find(
                address);
9           ...
10          return;
11      }
12      // Race. Try again, we now are not last
13  }
```

Fig. 10.5: Double-check mechanism with locking to prevent race conditions.

10.3.2 Access top bit

The access top bit is one of the most important features of the design, and the main reason just two data structures are needed. It keeps the information flow in a top-

down direction and eliminates the need to maintain a *Top Map*, which is a view on the current dependency graph from above. Nonetheless, its implementation is non-trivial.

Figure 10.6 shows a code snippet of the *DataAccess* structure internal implementation. Even though the top bit was described as a boolean, it is implemented as a *std::atomic<int>* [19]. This STL type will perform atomic increments and decrements, and allows the code to execute, in this case, an atomic *fetch_sub*.

```
1   struct DataAccess {
2   private:
3       //! The type of the access
4       DataAccessType _type;
5
6       //! Next task with an access matching this one
7       Task * _successor;
8
9       //! Is this the top read access
10      std::atomic<int> _isTop;
11      ...
12  public:
13      DataAccess(DataAccessType type, ...) : ..., _isTop(1)
14      { ... }
15
16      inline bool decreaseTop() {
17          int res = _isTop.fetch_sub(1, std::memory_order_relaxed);
18          assert(res >= 0);
19          return (res == 0);
20      }
21  }
```

Fig. 10.6: DataAccess structure and atomic top bit implementation.

Atomic decrements are more lightweight than locks, as no thread is blocked, accesses are just ordered according to cache coherency mechanisms [(1)]. In this case, it is used to implement a *decreaseTop()* call that returns true the second time it is called.

This fulfills the requirements for the top bit. It is only checked two times: when a read access finishes, to check if it is the top read, and when a top read is passing along its top bit.

In the code, the finishing task assumes it is the top read whenever this call returns true, as it means the *bit has been set to true*.

The implementation also assumes the next read has finished if decrementing its top counter returns true, as that would mean that access has already checked if it was the top read, and that is only done when finishing.

[(1)] This is true for the *fetch_sub* implementation x86-64 and similar architectures [20]

This mechanism is used to prevent race conditions. If this was a standard boolean, the value could be changed by one thread while the another has just checked the value and is about to change it.

That means as well that this counter is decremented as well when a read task is created and has no predecessor, because no task will decrement that counter otherwise, and the runtime would hang.

11 | Optimization

This chapter describes all the modifications that have been done to the *discrete-simple* dependencies after the initial implementation has been completed, both to enhance performance and add more functionality.

11.1 Allocating all the accesses with the task

On the initial implementation code, both the bottom map and the access array are dynamic structures. The bottom map is dynamic because there is no way of knowing how many accesses any of a tasks' children will have, and the access array is dynamic because the is no way to know how many accesses the current task will have at the moment it is being allocated, because they are created individually later, as explained in Subsection 8.3.2.

When thousands of tasks with many dependencies are being allocated, this can become a problem. Both structures used, a unordered_map and a vector, will allocate memory somewhere else and store a pointer to it. Then, if more elements are added, and no memory is left, a *resize* will happen, which will allocate a bigger chunk of memory and move every element there. This is a big performance penalty, and on very small granularity tasks can account for a sizeable amount of time. Figure 11.1 shows a diagram of where the access data is allocated in relation to the *TaskDataAccesses* struct.

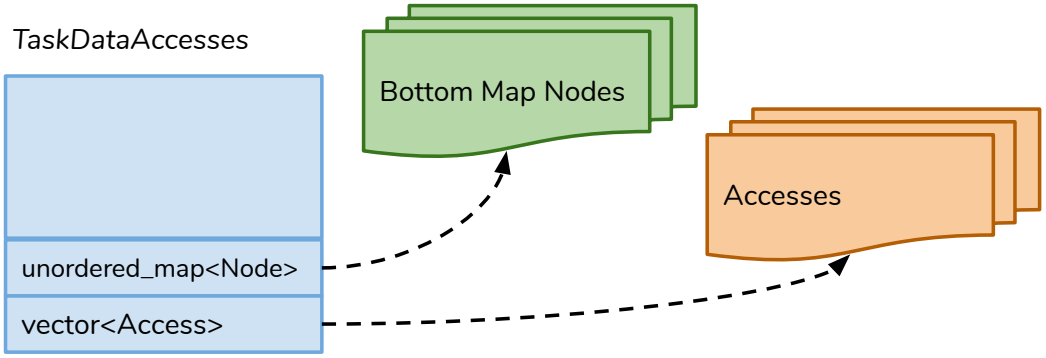


Fig. 11.1: Allocation of the *TaskDataAccesses* struct before optimization.

At this point, not enough information is available at the runtime to find a better solution to this problem. However, an API change to Nanos6 could be proposed, and implemented in the *Mercurium* compiler, to pass the number of accesses that a task has at the moment the runtime is allocating it, because the compiler has the ability to get that information at compile time (runtime in the case of multidependencies).

The current call that prompts the runtime to allocate space for a Task, and thus allocate the *TaskDataAccesses* struct, is *nanos6_create_task*, called from the annotated program, and the proposed change would be to add an extra parameter to send the number of accesses that are expected to register. Figure 11.2 shows the *nanos6_create_task* call with the new parameter.

```
1  void nanos6_create_task(
2      nanos6_task_info_t *task_info,
3      nanos6_task_invocation_info_t *task_invocation_info,
4      size_t args_block_size,
5      /* OUT */ void **args_block_pointer,
6      /* OUT */ void **task_pointer,
7      size_t flags,
8      /* NEW PARAMETER */ size_t num_deps
9  );
```

Fig. 11.2: Task creation API with proposed change to pass access number.

With the new parameter, the exact space needed for each task with its dependencies can be allocated, and only the bottom map is dynamic. This enables the runtime to prevent at least one and potentially more dynamic allocations when resizing the array.

Furthermore, it is possible to allocate the array of tasks with the same call to the *Memory Allocator* subsystem and have that array live in the bottom of the *TaskDataAccesses* struct, by modifying the array pointer to that address. Figure 11.3 shows the data allocation in memory, where it can be clearly seen that now the access array is part of the same chunk of memory than the *TaskDataAccesses* struct. This way, another allocation per task can be prevented.

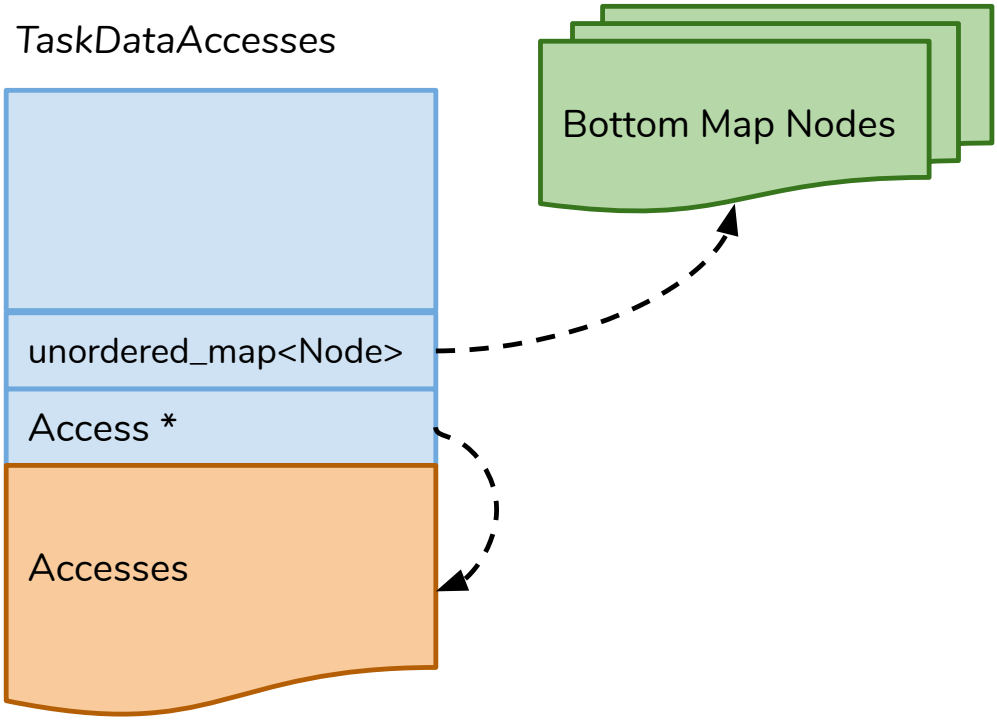


Fig. 11.3: Allocation of the *TaskDataAccesses* struct after optimization.

Figure 11.4 is a code snippet from the runtime entry point for the *nanos6_create_task* call, where tasks are allocated. There, the needed space for the dependencies is calculated based on the number of accesses.

```
1   #ifdef DISCRETE_SIMPLE_DEPS
2       // We use num_deps to create the correctly sized vector for storing
            the dependencies.
3
4       size_t seqsSize = sizeof(DataAccess) * num_deps;
5       size_t addrSize = sizeof(void *) * num_deps;
6   #else
7       size_t seqsSize = 0;
8       size_t addrSize = 0;
9   #endif
10
11      // Allocation and layout
12      *args_block_pointer = MemoryAllocator::alloc(args_block_size +
            taskSize + seqsSize + addrSize);
```

Fig. 11.4: Code snippet of the *nanos6_create_task* function showing adaptive memory allocation.

11.2 Reductions

11.2.1 Requirements

One very important feature of the OmpSs-2 task model [5] is the support for reductions in the task clause. The dependency subsystem is responsible for providing each thread with a private space to accumulate the results, and then combine all of them when the reduction has finished. As such, the runtime needs to keep track of each reduction, and manage its lifetime inside the data structures.

On a very high level, the reductions have the following peculiarities:

- Reductions can always start (never block the task) because they operate on private memory. This also means they execute concurrently.
- Reductions cannot be combined until they are closed and all accesses have finished.
- A reduction is closed when there are no more tasks of that same reduction accessing that address immediately after the last. That can happen if a non-reduction access is registered, or if a Taskwait happens on the parent.
- All other accesses have to block if a reduction precedes them.

```
1  int main() {
2      int A = 0;
3
4      for(int i = 0; i < 5; ++i) {
5          #pragma oss task reduction(+: A)
6          {
7              A += 1;
8          }
9      }
10
11     #pragma oss task in(A)
12     {
13         std::cout << A << std::endl;
14     }
15 }
```

Fig. 11.5: Sample code of a program that uses reductions

Figure 11.6 illustrates the concepts that have been explained, by representing the sample code of Figure 11.5. All accesses to the same address A will execute concurrently, and the reduction will be closed upon a different access or a taskwait (in our case, a different access). Note that the concept of *top* is also included in the figure, as it will be necessary for the implementation.

11.2.2 Implementation

Reductions are implemented on our version of the dependency subsystem very similarly to *read* accesses, but with some key differences. They share the same *top* bit logic explained on Section 10.1 and 10.3, but additional logic has to be added to manage not only the task lifespan but also the reduction lifespan, because all the reduction tasks might be deleted but until that reduction is closed (by another access or a taskwait), it cannot be combined and deallocated.

To achieve this, the task and reduction's lifespans will be tracked separately. Tasks will work just like read accesses, but reductions pending to be combined have to be checked every time a reduction task finishes, a taskwait is issued, or a new access is registered. Keeping track of what is the current reduction to be closed for each address will be done with the addition of a new field in the *BottomMapEntry* class, which represents a node of the bottom map. This field will contain the current reduction information for that address. With that, Figure 11.7 shows the structure of the *BottomMapEntry* class.

```
1  struct BottomMapEntry {
2      DataAccess * access;
3      bool satisfied;
4      ReductionInfo * reductionInfo;
5  };
```

Fig. 11.7: BottomMapEntry layout after implementing reductions.

Some more information is needed as well for each reduction, which is passed to the runtime when registering a reduction access. That information includes the reduction operator, and the length of the region that needs to be reserved for allocating the intermediate results. The extra fields will be saved in the *DataAccess* structure, for posterior use. In that same class a pointer to a new struct called *ReductionInfo* will also be saved. This structure includes all the specific information for that reduction sequence as well as counters to know if the reduction is finished.

With all the earlier changes, Figure 11.8 shows the layout of our DataAccess structure ready to store everything needed to implement reductions.

```
struct DataAccess {
    private:
        //! The type of the access
        DataAccessType _type;

        ...

        //! Reduction-specific information of current access
        ReductionInfo *_reductionInfo;

        bool _closesReduction;

        //! Reduction stuff
        size_t _reductionLength;

        reduction_type_and_operator_index_t _reductionOperator;

        //! Next task with an access matching this one
        Task * _successor;

        //! Is this the top read access..
        std::atomic<int> _isTop;

        ...
};
```

Fig. 11.8: DataAccess layout after implementing reductions.

To keep track when a reduction has finished, a similar technique as other places in the runtime is used with a *std::atomic<int>* in the reduction-specific structure, and functions that will return true to whoever is in charge of combining the reduction through the calls *ReductionInfo::incrementRegisteredAccesses()* and *ReductionInfo::incrementUnregisteredAccesses()*. This will be used to manage the reduction lifespan.

With all those details sorted and clear logic to follow, reduction support in the runtime can be implemented. It has to be noted that unlike normal data accesses, reductions will not be able to be nested with our implementation without modifications, as that would mean in essence implementing the *weakreductions* feature that the OmpSs-2 spec details.

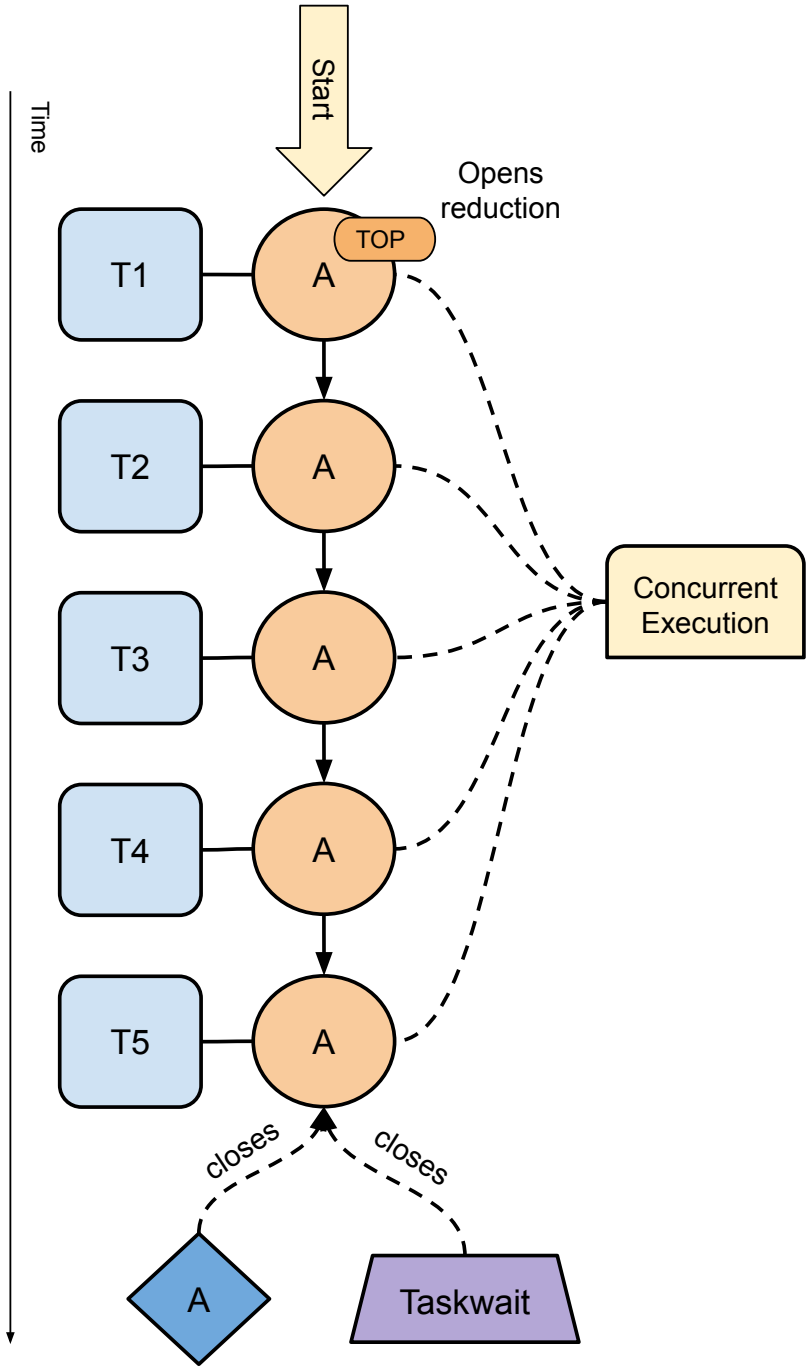


Fig. 11.6: High-level illustration of a task reduction

12 | Evaluation

This chapter dives into the experimental design used for evaluating the performance of the *discrete-simple* implementation and presents the result of those experiments.

12.1 Experimental design

The evaluation of the performance of the *discrete-simple* implementation versus the earlier *linear-regions-fragmented* implementation has been done in the CTE-KNL Supercomputer of the Barcelona Supercomputer Center. Each node of the supercomputer has an Intel® Xeon™ Phi processor (a x86 manycore) with 64 cores and configured without hyperthreading and in quadrant cluster mode. This processors, being based on Intel Atom architecture, normally exhibit low per-core performance but their core count will show better the scalability of the implementations, which is the determining factor [21].

Several benchmarks, detailed in Section 12.2, will be run on individual nodes of the supercomputer. All the benchmarks are programmed adhering to the OmpSs-2 Spec [5], they make use of data dependencies and are designed to be a representative example or real-world workloads. They were not created for this project specially, but rather are already available on the BSC and used to monitor the performance of the runtime across different versions.

The benchmarks will be run with a fixed problem size and varying block sizes, serving the purpose of evaluating the scalability of each implementation. It is expected that for an implementation to be considered better in performance, it should show greater performance in small block sizes. For very big block sizes, as very few tasks will be issued and the benchmarks will run out of parallelism, not much difference in performance should exist.

Finally, Table 12.1 shows the different versions of software that were present in the CTE-KNL supercomputer during the tests. This is important because results of this benchmarks might change if done under different versions of this software.

Software	Version
OmpSs-2	git (1b94838300a8)
GNU Compiler Collection	6.3.0
Boost	1.63.0
Intel® MKL (Cholesky)	2017.0002
Linux Kernel	SUSE 4.4.21-69

Table 12.1: Software versions present on the CTE-KNL supercomputer during the final evaluation

12.2 Benchmarks

All the results of the benchmarks are based on the average of thirty executions, having had a warm-up execution before. The code inside every benchmark would do the following:

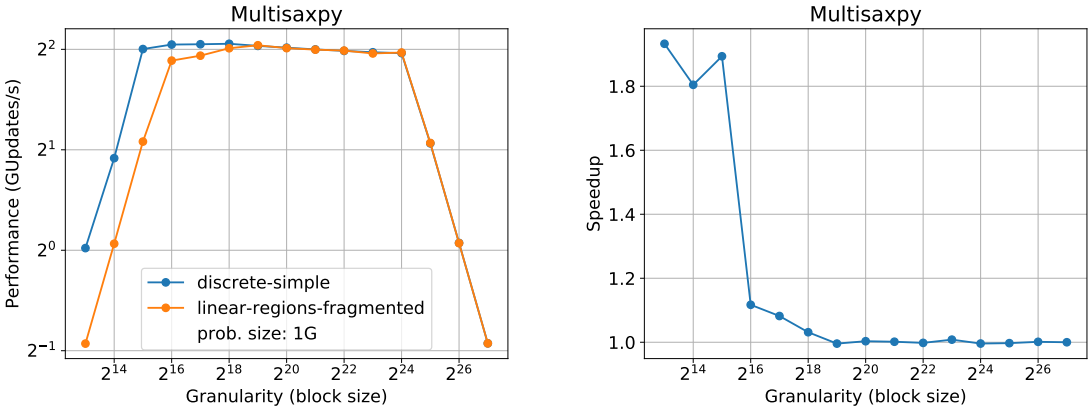
1. Execute the benchmark once as a warm-up.
2. Start the timing.
3. Execute thirty runs of the benchmark.
4. Stop timing and divide the result by thirty.

12.2.1 Multisaxpy

A SAXPY operation is a *Single-Precision A*X Plus Y*, which is a simple function, present in the standard *Basic Linear Algebra Subroutines* [22] which combines scalar multiplication and vector addition. It takes two vectors **X** and **Y**, and a scalar **a** and performs the following operation for each element:

$$Y_n = a \cdot X_n + Y_n$$

The Multisaxpy variant developed in the BSC is only a variant that performs multiple iterations of the SAXPY operation on a very large vector, to create a benchmark that represents extremely memory-bound applications.



(a) Scalability (b) Speedup

Fig. 12.1: Scalability and speedup plots of the Multisaxpy benchmark with a problem size of 1G elements

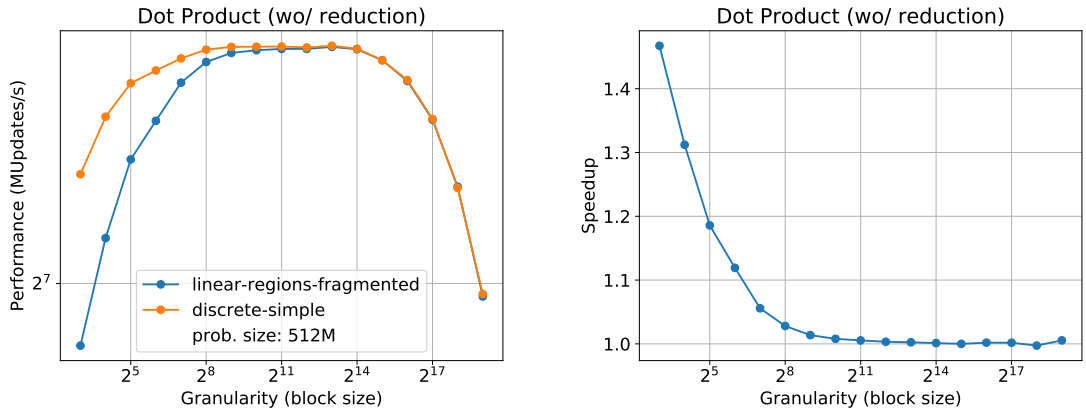
Figure 12.1 shows the result of the Multisaxpy benchmark for both dependency implementations. It is clear that the *discrete-simple* implementation outperforms the existing one by a wide margin in scalability, having much better performance for small block sizes and reaching peak performance for three runs more than *linear-regions-fragmented*. It also shows almost a $2X$ speedup for those small chunks.

12.2.2 Dot product

In linear algebra, the dot product of two equal-length vectors **a** and **b** is defined as the sum of the products of each element of the vectors:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n} a_i \cdot b_i$$

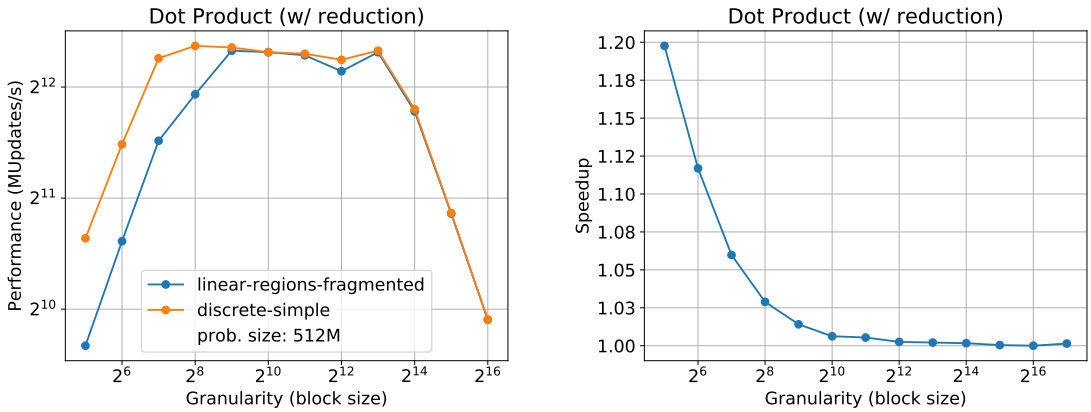
The dot product benchmark does so with very long vectors, simulating in a similar way to Multisaxpy a memory-bound application. Moreover, the BSC version of the dot product benchmark has two different implementations that are interesting for the dependency subsystem: a simple one with normal dependencies and one with reductions. This will allow to evaluate the reduction implementation in *discrete-simple* that was described in Section 11.2.



(a) Scalability (b) Speedup

Fig. 12.2: Scalability and speedup plots of the Dot Product benchmark without using reduction and a problem size of 512M elements

Figure 12.2 shows the scalability of the dot-product in the regular dependency version (without reduction). It is clear that the overhead reduction of the *discrete-simple* implementation in this case translates to better scalability and earlier reach of peak performance, measurable speedups over the *linear-regions-fragmented* implementation.



(a) Scalability (b) Speedup

Fig. 12.3: Scalability and speedup plots of the Dot Product benchmark using reduction and a problem size of 512M elements

Figure 12.3 shows the same benchmark but implemented using the reduction feature of OmpSs-2. Again, the *discrete-simple* implementation shows better scaling, achieving peak performance with smaller block sizes and $1.2X$ speedups on the smaller granularities.

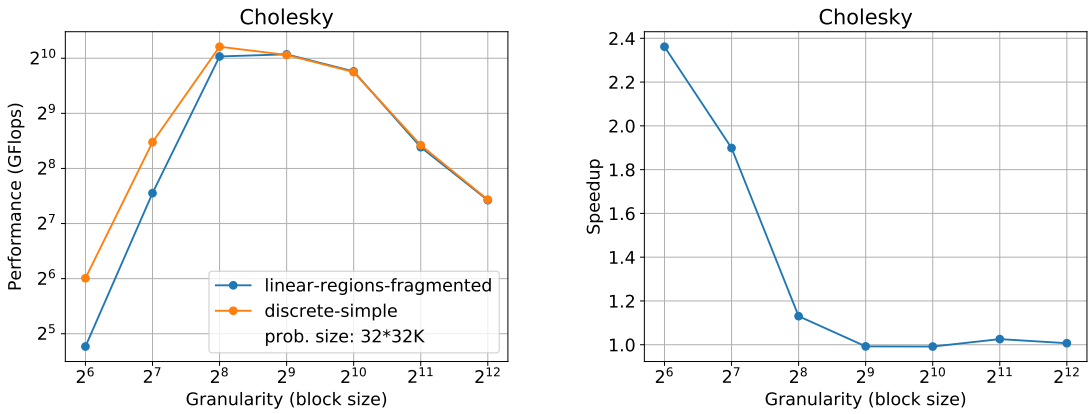
12.2.3 Cholesky

The Cholesky algorithm is used to decompose a Hermitian positive-definite matrix **A** into the product of the a lower triangular matrix **L** and its conjugate transpose **L***, resulting in the following equality:

$$A = L \cdot L^*$$

The version of the Cholesky decomposition used as a benchmark in the BSC uses the provided functions on the *BLAS* and *LAPACK* libraries [22, 23] to calculate the decomposed matrices. In the CTE-KNL, the implementation of such libraries that has been used is the Intel® MKL (Math Kernel Library) [24].

This benchmark is used to simulate a CPU-bound application, in which most of the work is done on computation rather than memory access. This is why problem sizes are smaller as well.



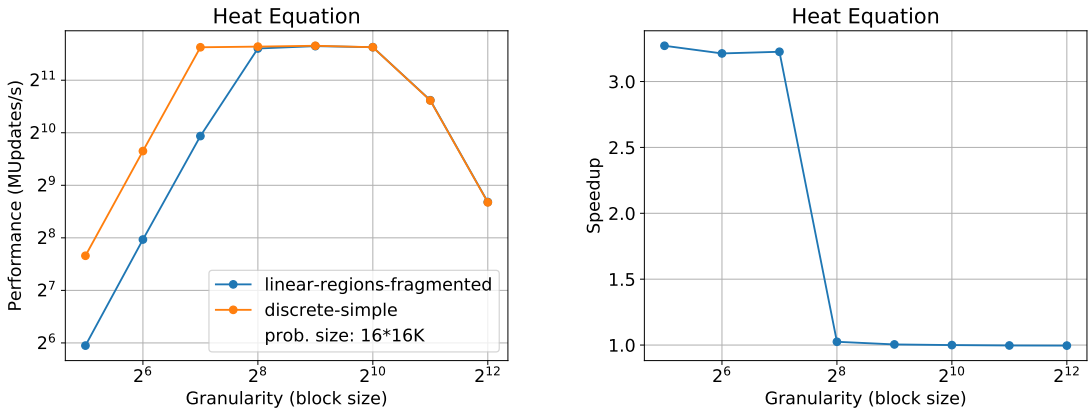
(a) Scalability (b) Speedup

Fig. 12.4: Scalability and speedup plots of the Cholesky benchmark with a problem size of 32*32K elements

In Figure 12.4, it is clear that even on CPU-bound applications, the *discrete-simple* implementation provides better than the *linear-regions-fragmented*, with more than a $2X$ speedup on low block sizes and achieving again for this benchmark a better potential peak performance for a block size of 2^8.

12.2.4 Heat Equation

The heat equation is a parabolic partial differential equation that describes the distribution of heat in a region over time. It has important real-life usages like the study of Brownian Motion and chemical diffusion. The BSC benchmark that implements the heat equation solves it using an iterative Gauss-Seidel method [25]. It provides a good example of a CPU-bound application that does iterative computations in a matrix.



(a) Scalability (b) Speedup

Fig. 12.5: Scalability and speedup plots of the Heat Equation benchmark with a problem size of 16*16K elements

Figure 12.5 shows the scalability plot for the benchmark. Very similarly to other benchmarks, the *discrete-simple* implementation comes ahead on scalability, reaching again peak performance with smaller block sizes, and achieving more than 3X speedups in some cases.

12.2.5 Matrix Multiply

A classic case study in parallelism is the performance of matrix multiplication. Instead of using a simple non-parallelizable iterative matrix multiply, we can split matrices into blocks and use a blocked matrix multiply algorithm [25]. This can only be done with matrices that fit certain constraints.

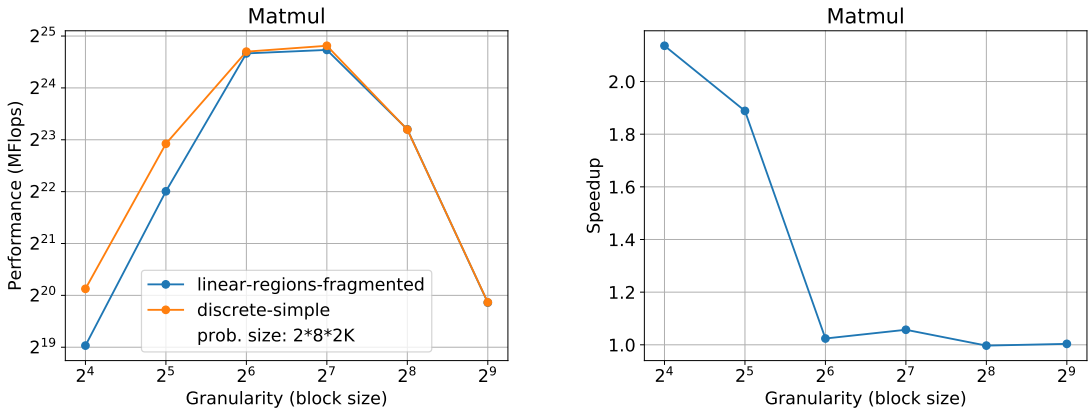
For instance if we have two matrices $A_{(m,n)}$ and $B_{(n,p)}$, a block size that divides all sizes and we partition both matrices by the given block sizes, we can express the following product

$$C_{(m,p)} = AB$$

Using only the partitions

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Those are the mathematical grounds for the benchmark of matrix multiply chosen in this project. Using the different blocks, we can express the multiplication through task dependencies between blocks of memory and the dependency subsystem can discover the parallelism for those operations at runtime. This is generally a CPU-bound benchmark.



(a) Scalability (b) Speedup

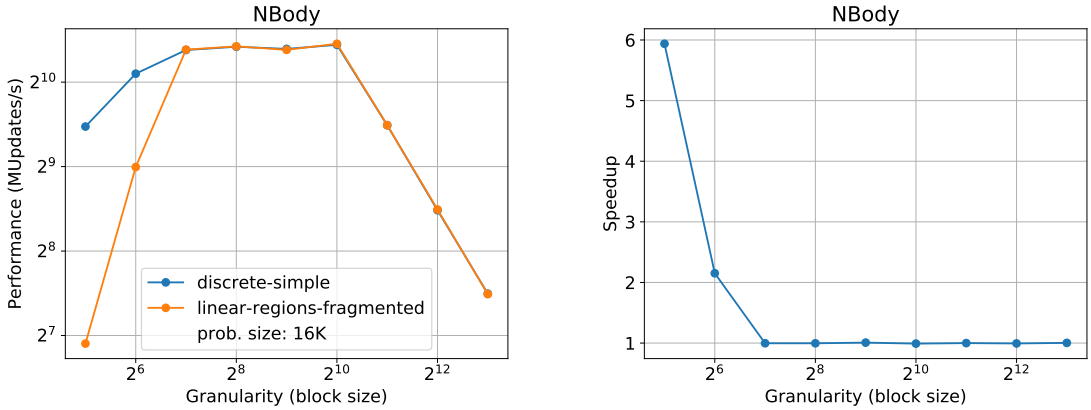
Fig. 12.6: Scalability and speedup plots of the Matrix Multiply benchmark with a problem size of 2*8*2K elements

Figure 12.6 is the scalability plot for matrix multiply on the CTE-KNL supercomputer. We observe $2X$ speedups for low granularities, and in this case even some slight speedup during peak performance. It is, however, different to some other benchmarks in the sense that the *discrete-simple* implementation does not reach the best performance before than the *linear-regions-fragmented* implementation. As this is similar to the Cholesky benchmark, it is a clear example of how different workloads produce very different speedups with each dependency subsystem, and the importance of having a great variety of benchmarks.

12.2.6 N-body

The N-body benchmark is a simulation that approximates the evolution of a system of bodies in which each body interacts with every other body. For example, this could be used as an astrophysical simulation in which each body is a star, and all the bodies attract each other through gravity.

N-body simulation is used in several computational science problems, such as protein folding, fluid flow simulation, and global illumination in computer graphics. In this case, the BSC benchmark uses the aforementioned gravity body example, and simulates a number of particles through a number of time steps.

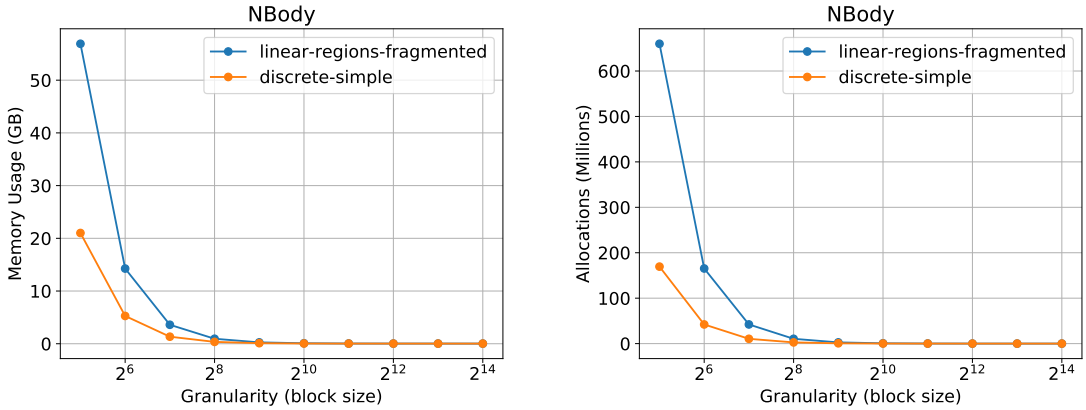


(a) Scalability (b) Speedup

Fig. 12.7: Scalability and speedup plots of the N-body benchmark with a problem size of 16K particles and 10 time steps

Figure 12.7 is the scalability plot of the simulation in the CTE-KNL supercomputer. It shows an unusually high speedup in the first data point, and in the case of this benchmark that is due to the high memory usage of the *linear-regions-fragmented* implementation. High memory usages pose a problem because the memory allocator has a harder time finding suitable chunks of memory for the program and slows down the execution greatly. This can also cause the node to go into *thrashing*, slowing every program running on it exponentially.

To better illustrate this point, it is possible to collect the number of allocations and the peak memory usage (resident set). Figure 12.8 shows plots corresponding to those metrics. It is clear that for high granularities the maximum memory used by the program spikes, but the *discrete-simple* implementation will use close to a third for small block sizes in this case. In fact, it was not possible to collect metrics for smaller block sizes due to the *linear-regions-fragmented* version getting killed by the system due to high memory usage.



(a) Memory Usage (b) Allocations

Fig. 12.8: Peak memory usage and total allocations of the N-body benchmark with a problem size of 16K particles and 10 time steps

Allocations, also shown in Figure 12.8, show a similar story. The number of memory allocations (calls to *malloc*, *realloc* and *memalign*) spikes for low granularities in an exponential fashion, reaching almost 700 million during the execution of the NBody benchmark with the *linear-regions-fragmanted* implementation.

The memory usage explains really well the performance degradation in Figure 12.7. In the case of the CTE-KNL, each node has 16GB of high bandwidth memory that is configured in a cache mode. This means that programs only using less than 16GB of memory will be fully cached in the faster memory and thus exceeding that amount will penalize heavily the program. This accounts for the higher than normal speedup in the smallest granularity.

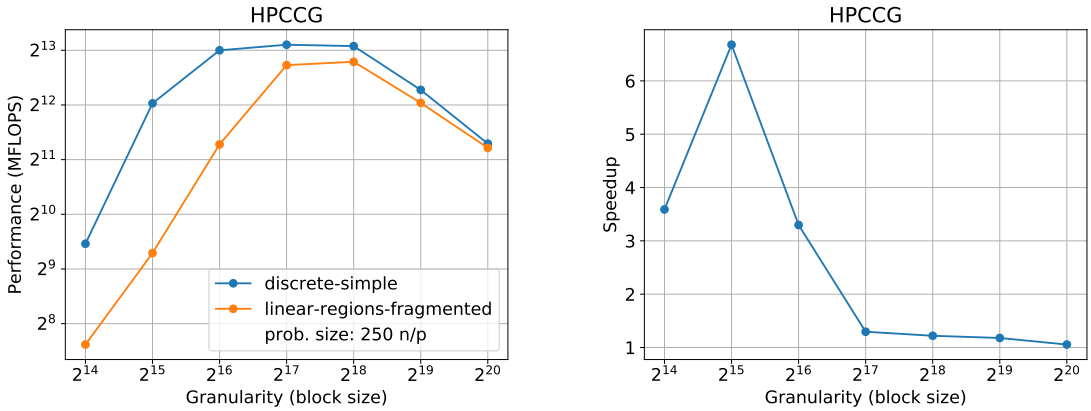
This is another of the benefits of the new *discrete-simple* implementation. It does less allocations (thus having less memory fragmentation) and uses less memory overall, reducing overhead due to memory usage and leaving more resources for the actual program to consume. In the specific case of the NBody benchmark, the *discrete-simple* implementation averaged 4 memory allocations per each task created, while the *linear-regions-fragmented* did an average of 15 allocations per task.

12.2.7 HPCCG

The HPCCG benchmark stands for High Performance Computing Conjugate Gradients [26] and is a simple and open-sourced conjugate gradient benchmark. Conjugate Gradients are methods to solve n linear equations with n unknowns, particularly targeted at cases where that n is big [27]. In the particular case of our benchmark, the implementation is designed to be very scalable, and is a great test case mixing a lot of tasks with discrete dependencies and several reductions on different variables.

In real world applications, the Conjugate Gradients method is used for solving linear equations on systems too big for other algorithms such as the Cholesky decomposition

we described in Subsection 12.2.3, and can be used for optimization problems.



(a) Scalability (b) Speedup

Fig. 12.9: Scalability and speedup plots of the HPCCG benchmark with a problem size of 250 nodes/processor

Figure 12.9 shows that similarly to other benchmarks, the *discrete-simple* implementation can have speedups of more than $6X$ on small granularities, and finally the difference levels out when the block size is bigger than ideal. This is a great example of a use case where the decision to use *discrete-simple* dependencies would prove beneficial for the user regardless of the chosen block size.

This is explained because apparently the new implementation does not only provide better scaling, but in some cases reduces the per-dependency creation overhead enough to see a significant speedup. This will only happen, however, on benchmarks where dependency creation accounted for a big enough portion of the total execution time (which is not common), as per Amdahl's Law [7].

13 | Conclusion

This project has introduced a new dependency subsystem implementation to the Nanos6 runtime for the OmpSs-2 programming model. This implementation has been tested through a series of programs prepared during the runtime analysis and by the benchmarks done in the evaluation phase. Using different dependency subsystems is also easier now, because the user can switch between them using environment variables without the need of recompiling the full runtime.

The new dependency subsystem has also been optimized and enhanced through the addition of elements to the API with the compiler that allow for more effective memory allocation and support for reductions.

It has been proved through experimentation with help of the hardware resources at the BSC that the new implementation can perform and scale better in a variety of scenarios, in both CPU-bound and memory-bound high performance applications.

This enhancement will be incorporated in a future release of OmpSs-2 and users of the runtime that use only dependency features that are included in the *discrete-simple* implementation will be able to switch to it for a potential speedup. With this, all the requirements specified in Section 3.2 have been fulfilled.

For the student, this project has been an invaluable learning experience that has led him to apply techniques, algorithms and knowledge in general that was acquired during the Informatics Engineering degree, as well as to acquire new skills and even more knowledge.

14 | Future Work

This chapter covers possible future work that may be done on the runtime based on the changes made by this project, but that either was not on the scope or there wasn't enough time to implement.

First of all, the next logical step would be to add support in the *discrete-simple* implementation for the *weakdepend* clause, as it is estimated that can be incorporated without causing a significant amount of overhead and would increment the use cases that can be handled. This would include possibly as well support for *weakreductions* (nested reductions).

Secondly, this work could be integrated with other research currently ongoing at the BSC that is centered on providing hints to the runtime, either coming from the programming or a subsystem that recollects statistical data based on passed iterations, that could be used by the dependency subsystem to apply optimizations for certain usage patterns that would allow for significant performance increments. One example of this would be to hold off child task execution if the parent task is a simple task-creation loop, to prevent locking between child and parent tasks.

Finally, another possible optimization to dampen dependency creation overhead would be to offload task dependency registration to a worker thread, so that the thread creating the tasks doesn't get blocked while the subsystem is registering dependencies.

Bibliography

[1] Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, Jan 2018. doi: 10.1109/IEEESTD.2018.8277153. 1

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. 1, 3

[3] Intel Corporation. Threading Building Blocks, . URL https://www.threadingbuildingblocks.org/. Accessed: 2019-06-03. 3

[4] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 5.0, November 2018. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accessed: 2019-05-14. 3, 6

[5] Barcelona Supercomputing Center. OmpSs-2 Specification, . URL https://pm.bsc.es/ompss-2-docs/spec/. Accessed: 2019-05-07. 1, 4, 6, 7, 27, 29, 30, 33, 35, 41, 50, 54

[6] Barcelona Supercomputing Center - Programming Models. Nanos6 GitHub Mirror. URL https://github.com/bsc-pm/nanos6. Accessed: 2019-05-07. 2

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. 3, 63

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Nov 1994. doi: 10.1109/SFCS.1994.365680. 3

[9] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface 1.0, October 1997. URL https://www.openmp.org/wp-content/uploads/fspec10.pdf. Accessed: 2019-05-07. 3

[10] OpenMP Architecture Review Board. OpenMP Compilers & Tools. URL https://www.openmp.org/resources/openmp-compilers-tools/. Accessed: 2019-05-16. 4

[11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728. 5

[12] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. Improving the integration of task nesting and dependencies in openmp. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 809–818, May 2017. doi: 10.1109/IPDPS.2017.69. 6

[13] Ferran Pallarès Roca. Extending ompss programming model with task reductions: A compiler and runtime approach. Bachelor's thesis, Barcelona School of Informatics, Universitat Politècnica de Catalunya, 1 2017. 7

[14] Kent Beck. *Test Driven Development. By Example*. Addison-Wesley Longman, Amsterdam, 2002. ISBN 0321146530. 10

[15] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445. 12

[16] GNU Project. Automake. URL https://www.gnu.org/software/automake/. Accessed: 2019-05-11. 29

[17] Barcelona Supercomputing Center. Extrae, . URL https://tools.bsc.es/extrae. Accessed: 2019-05-20. 31

[18] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008. ISBN 9780471455936. 33

[19] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. 44, 45

[20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, May 2019. 45

[21] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016. ISSN 0272-1732. doi: 10.1109/MM.2016.25. 54

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. ISSN 0098-3500. doi: 10.1145/355841.355847. 55, 58

[23] LAPACK Team. LAPACK - Linear Algebra PACKage. URL http://www.netlib.org/lapack/. Accessed: 2019-05-29. 58

[24] Intel Corporation. Intel® MKL - Math Kernel Library, . URL https://software.intel.com/en-us/mkl/. Accessed: 2019-05-29. 58

[25] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3. ed.)*. Johns Hopkins University Press, 1996. ISBN 978-0-8018-5414-9. 58, 59

[26] Michael A. Heroux. HPCCG: High Performance Computing Conjugate Gradients. URL https://github.com/Mantevo/HPCCG. Accessed: 2019-06-08. 62

[27] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J Res NIST*, 49(6):409–436, 1952. doi: 10.6028/jres.049.044. 62